Abstract

This thesis presents a software development project in the fields of generative programming and object oriented distributed systems with focus on the Java 2 Enterprise Edition (J2EE) standard. The program developed in this project is called 3E and is an open source command line tool for generating Enterprise JavaBeans. By providing 3E with an XML document specifying an application specification 3E will parse the XML, perform multiple checks on the input data to ensure that the EJB2.0 specification is observed and finally generate the files for a complete Enterprise JavaBean application. An additional feature of 3E is to generate the XML input file specifying an Enterprise JavaBean application reflecting the design of tables and relations in a database.

3E is a Java program integrated with Ant. 3E uses Castor XML to parse XML and the open source template language Velocity to generate code.

Contents

Foreword & Preface **5**

1 Background analysis **7**
1.1 J2EE and the EJB2.0 specification . . . 7
1.1.1 Enterprise JavaBeans 2.0 . . . 7
1.1.2 The elements of an EJB component . . . 8
1.1.3 The EJB2.0 specification for CMP Entity Beans . . . 9
1.2 Generative programming and code generation . . . 11
1.2.1 Generators . . . 11
1.2.2 Code generators . . . 12
1.2.3 Why develop code generators? . . . 12
1.3 Applying generative programming to the EJB2.0 domain . . . 13

2 Method **15**
2.1 An open source project based on open source projects . . . 15
2.2 Development environment . . . 15
2.3 Development process . . . 16
2.4 Terminology . . . 16

3 Analysis of requirements **17**
3.1 EJB generators . . . 17
3.1.1 XDoclets . . . 17
3.1.2 MiddleGen . . . 18
3.1.3 EJBBuilder . . . 18
3.1.4 EJBGen . . . 19
3.1.5 JDeveloper . . . 19
3.1.6 TEGA – The EJB Generator Application . . . 19
3.1.7 Requirements . . . 20
3.2 Requirements analysis . . . 20
3.2.1 Interview with an EJB developer . . . 20

4 Design **23**
4.1 Program structure . . . 23
4.2 The application specification . . . 25
4.2.1 Selecting an input format . . . 25
4.2.2 A language for the application specification . . . 25
4.3 Validating the application specification . . . 27
4.4 Collect information from application specification . . . 27

4.4.1 XML Parsers . . . 28
4.4.2 Some options . . . 28
4.4.3 Castor XML . . . 29
4.5 The application model . . . 32
4.6 How to process information in the application model . . . 33
4.6.1 The Visitor Pattern . . . 34
4.7 Generating Java and XML from the application model . . . 35
4.7.1 JSP and Tag Libraries . . . 36
4.7.2 Velocity . . . 37
4.7.3 Choosing a code generation technology . . . 39
4.7.4 More about how Velocity works . . . 39
4.8 Design of specific features . . . 40
4.8.1 Automatic generation of primary keys . . . 40
4.8.2 Supporting compound keys . . . 41
4.8.3 Representing relations between Entity Beans . . . 42
4.8.4 Generating debug messages . . . 43
4.8.5 Read from database . . . 44

5 Installation and user manual **46**
5.1 What is 3E? . . . 46
5.2 How to use 3E . . . 47
5.2.1 Specifying an example input file . . . 47
5.2.2 Using Ant . . . 48
5.3 Installing 3E . . . 49
5.3.1 System requirements . . . 49
5.3.2 Installing 3E . . . 49
5.4 Developer's guide . . . 50
5.4.1 Elements in the XML input file . . . 50
5.4.2 Exceptions when running 3E . . . 56
5.4.3 Specifying an XML input file for database connection . . . 58

6 Technical description **59**
6.1 Structure of 3E source code . . . 59
6.2 XML Schema for an application specification expressed in XML . . . 61
6.3 Mapping files needed by Castor XML . . . 62
6.4 Implementation of the application model . . . 63
6.5 Visitors for checking and completing the application model . . . 64
6.5.1 The five concrete visitor classes . . . 64
6.5.2 The static helper methods . . . 66
6.5.3 Possible improvements and extensions . . . 67
6.6 Velocity Templates used for code generation . . . 67
6.6.1 The Velocity language . . . 67
6.6.2 Template layout and formatting of generated output . . . 68
6.6.3 Velocity macros . . . 69
6.6.4 The Utility class . . . 69
6.6.5 How automatic generation of primary keys is implemented . . . 69
6.6.6 How compound keys are generated . . . 70
6.6.7 How to create a new template . . . 71

6.7 Generating debug messages . . . 71
6.7.1 The Unmarshaller object . . . 71
6.7.2 The BogmUnmarshaller . . . 72
6.7.3 Writing debug messages . . . 74
6.7.4 Improvements . . . 74
6.8 Exception handling . . . 75
6.8.1 Possible improvements . . . 75
6.9 Generating an input file for 3E from database . . . 76
6.9.1 DatabaseMetadata objects . . . 76
6.9.2 SchemaFactory . . . 76
6.9.3 Type conversion — from SQL types to Java types . . . 76

7 Test **78**
7.1 Test of checks made by 3E . . . 78
7.2 Test case . . . 79
7.2.1 Creating the XML input file . . . 80
7.2.2 Testing the generated EJB application . . . 81
7.2.3 Transactions and rollback . . . 83
7.2.4 Compound key . . . 83
7.3 Conclusion on the test of 3E . . . 83

8 Conclusion **84**
8.1 3E . . . 84
8.1.1 Discussion and future work . . . 85
8.2 The development process . . . 86

Bibliography **88**

A Generating Java code and XML from JSP **90**

B Files generated by the input file Simple.xml **92**
B.1 Address.java . . . 93
B.2 AddressHome.java . . . 93
B.3 AddressBean.java . . . 94
B.4 AddressPK.java . . . 95
B.5 Customer.java . . . 96
B.6 CustomerHome.java . . . 96
B.7 CustomerBean.java . . . 97
B.8 GUIDProvider.java . . . 98
B.9 ejb-jar.xml . . . 100

C Test of checks done by visitors in 3E **102**

D The XML input file for the Titan business test case **114**

E JSP clients used in the Titan business test case **115**

F 3E source code **120**

Foreword

This report is a dissertation paper after a two years Masters education in the field of software development at the IT University of Copenhagen. The work has been conducted from September 2002 till February 2003 under supervision of Peter Sestoft, associate professor at The Royal Veterinary and Agricultural University and the IT University of Copenhagen and secondary supervisor Rasmus Lund, external lecturer at the IT University of Copenhagen.

We would like to thank: Peter Sestoft for his engagement and thorough supervision, Rasmus Lund for providing us with the idea of developing a code generator and for his enthusiasm toward the project and Morten Madsen for his comments, feedback, his engagement in the development process and for his insightful opinions in the world of Enterprise JavaBean development.

Preface

This thesis is a software development project in the fields of object oriented distributed systems and generative programming. The main motivation for making this project has been to improve our programming skills and to gain experience with completing a larger software development project, from requirements analysis, architectural design to implementation and test.

In the project we have developed a code generator tool for generating Enterprise JavaBeans. The idea to develop a code generator tool was presented to us by our secondary supervisor Rasmus Lund. In a preliminary project composed in May 2002, TEGA, a prototype of a code generator tool developed by Rasmus Lund was examined carefully, to provide us with insight and knowledge of how a code generator could be implemented [19].

The main purpose of the developed program is to increase productivity when developing J2EE applications. Secondly it will help developers produce code that conforms to the EJB2.0 specification, and spare them the tedious and repetitious work which it is to write Enterprise JavaBeans. Thirdly the program is considered as an extendable code generator tool, where changes in specifications and new features can easily be added. This is possible due to the architectural design and choice of applied technologies.

The tool developed in this project is called 3E, an abbreviation for "The Entity EJB Engine". 3E is an open source command line tool and its users are thought to be Java developers familiar with the J2EE domain. By providing 3E with an XML input file specifying an application specification the tool will parse the XML file, perform multiple checks of the input data to ensure that the EJB2.0 specification is observed and thirdly generate the files of a complete EJB application. An additional feature is to generate an XML input file reflecting tables and relations of a running database.

3E is distributed as a jar file including source code, class files, Javadoc and examples. 3E can be downloaded at `http://www.it-c.dk/people/anne/3E`

Chapter 1

Background analysis

This chapter presents the problem domain and consists of three sections. Section 1.1 is an introduction to J2EE and will briefly cover the architecture of J2EE applications. Furthermore the section will introduce the concepts of Enterprise JavaBeans and in particular cover the basic parts and requirements for an Entity Enterprise JavaBean. The purpose of Section 1.1 is to show how cumbersome developing Entity Enterprise JavaBeans can be.
Section 1.2 introduces the concept of generative programming and code generators. And finally in Section 1.3 we discuss applying generative programming to the Enterprise JavaBean domain.

1.1 J2EE and the EJB2.0 specification

Sun MicroSystems has developed a standard for object oriented distributed systems called J2EE (Java 2 Enterprise Edition). The *J2EE standard* is a specification for an application server, implemented by various *vendors*, such as BEA, IBM and Oracle. Closely associated with the J2EE standard is the *EJB2.0 (Enterprise JavaBean) standard*, a specification of how to develop Enterprise JavaBeans for J2EE application servers. The overall architecture of an J2EE application is illustrated in Figure 1.1

Figure 1.1 shows the multi-tiered architecture of an J2EE application. As illustrated, thin clients such as web browsers and mobile devices invoke servlets or Java Server Pages on a web server. Other kinds of clients communicate directly through RMI with the Enterprise JavaBeans located in the EJB container on the J2EE server. As illustrated, the contents of the EJB container is divided into two layers: A layer responsible for domain logic and a layer maintaining domain data. The final tier in the architecture is the data layer, represented here as database systems where domain data is stored.

1.1.1 Enterprise JavaBeans 2.0

The EJB2.0 specification specifies three different kinds of Enterprise JavaBeans as part of a multi-tiered architecture of an J2EE application.

Session Beans: Session Beans provide an object oriented view of the *domain logic* of an application. There are two kinds of Session Beans, statefull and stateless. Both statefull and stateless Session Beans can provide local and remote access. Session Beans are normally invoked by clients and work as a facade to Entity Beans.

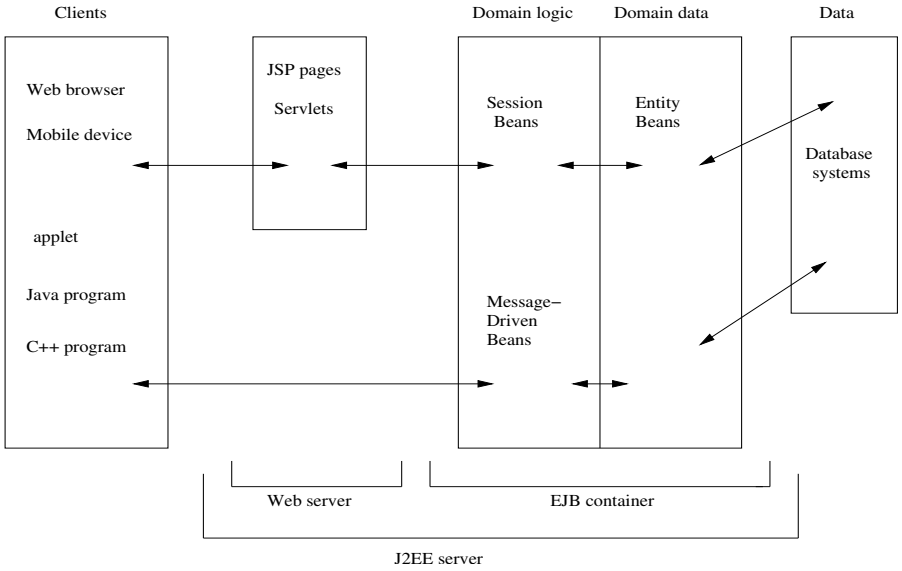

Figure 1.1: The architecture of J2EE applications

Entity Beans Entity Beans provide an object oriented view of the *domain data*. They work as a facade to the relational data in databases. There are two kinds of Entity Beans, BMP and CMP. In *BMP (Bean Managed Persistence)* the bean developer has the responsibility for implementation of all communication with the database. On the other hand, in *CMP (Container Managed Persistence)* the container implements the persistence mechanism. Both kinds of Entity Beans can provide local as well as remote access.

Message-Driven Beans Message-Driven Beans provide an asynchronous communication to domain logic.

1.1.2 The elements of an EJB component

An *EJB application* consists of different EJB components, where an *EJB component* is a single Session Bean, Message-Driven Bean or an Entity Bean. An EJB component consists of different elements, for instance, an EJB component being a CMP Entity Bean would consist of the elements shown in Figure 1.2.

Figure 1.2 illustrates how a CMP Entity Bean consists of an abstract *bean class*, *local and remote home interfaces* and *local and remote interfaces (also known as component interfaces)*. The different kinds of interfaces are optional, leaving the developer with the option of not allowing remote access to the Entity Bean by not defining remote interfaces and so forth. Furthermore there is a *deployment descriptor* which is an XML document describing all the EJB components and their relations in the EJB application. Besides the EJB application clients using the EJB components in the application can be specified. An EJB application together with clients are known as a J2EE application. When deploying a J2EE application on a J2EE server, all java files in the EJB application must be packed in a jar file. All client side files (Java server pages) must be packed in a war (web archive) file. The war file, the jar file and the deployment decriptor must be packed in an ear (EJB archive) file. The implementation of the abstract bean class and interfaces is generated by the EJB container at deployment-time.

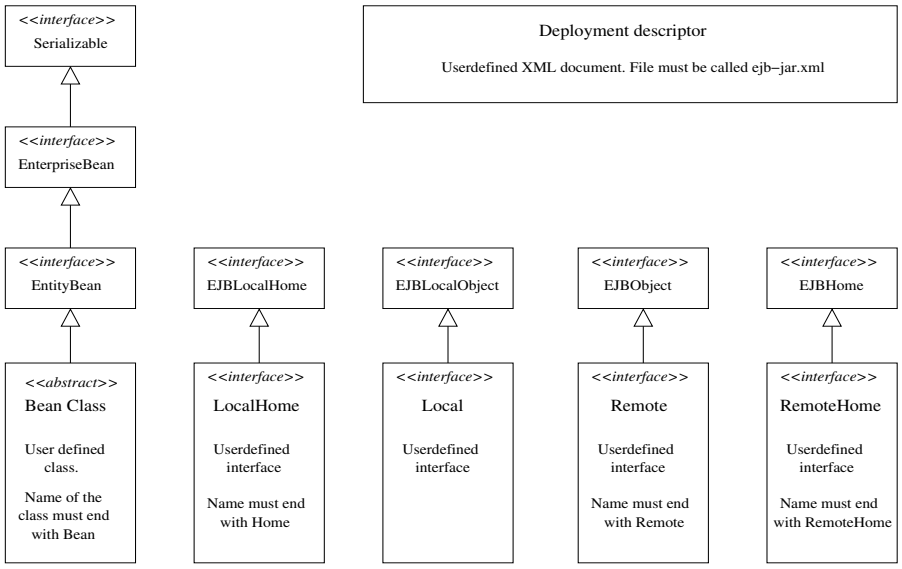

Figure 1.2: The elements of a CMP Entity Bean

1.1.3 The EJB2.0 specification for CMP Entity Beans

As mentioned, Entity Beans work as a facade to domain data in an EJB application and provide an object oriented view of relational data. As this thesis will focus on CMP Entity Beans (in the future referred to as Entity Beans) only the part of the EJB2.0 specification about CMP Entity Beans will be presented.

An Entity Bean must conform to the requirements of the EJB2.0 specification in order to be deployed in an EJB container. An Entity Bean consists of an abstract bean class and several interfaces, see Figure 1.2. Clients communicate with the bean class through the interfaces, never directly with the bean class. The *home interfaces* are used to define ordinary static methods while the *component interfaces* define ordinary non-static methods. The bean class does not implement the interfaces, but the bean class must behave nearly as if it did. Thus it is required to implement allmost every method specified in the interfaces, giving them slightly different names, see the method descriptions below. The interfaces are instead implemented by the implementation of the abstract bean class generated by the EJB container at deployment-time.

Declaring fields in an Entity Bean is not allowed. CMP fields are always defined by abstract get and set methods. At least one field must be specified as *primary key* in the deployment descriptor. Note that the primary key field cannot be of a primitive type, unless it is part of a compound key.

Entity Beans can form relations. There are three ways Entity Beans can relate to each other: one-to-one, one-to-many and many-to-many. Furthermore all three types of relations can be either unidirectional or bidirectional, leaving a total of 7 different relationship types between Entity Beans. In a bidirectional relation both Entity Beans can navigate to the relating Entity Bean through a *CMR (container managed relationship)* field. In unidirectional relationship only one of the involved Entity Beans has a CMR field, permitting navigation between the two Entity Beans in one direction only[15, 18]. Relations are described in the deployment descriptor and like the CMP fields, also CMR fields are declared by abstract get and set methods in the bean class.

Methods for an Entity Bean can be implemented or specified as abstract. Several standard methods have to be implemented in the bean class, thus a list of 7 standard callback methods used by the EJB container must all appear. The remaining methods are optional and are briefly described [15, 18]:

Query methods Find and Select methods are query methods. These kinds of methods are characterized by the use of EJB QL. *EJB QL* is the query language specified in the EJB2.0 specification, and is written as part of the query method description in the deployment descriptor.

Find methods May not be declared in the bean class, but are only declared in the home interfaces and described in the deployment descriptor. All Entity Beans must specify the findByPrimaryKey method in the home interface but may not implement it in the bean class. As a naming convention all find methods are always named find<method-name-suffix>

Select methods Are only used by the bean class. They may not be exposed in the interfaces and are declared as abstract methods in the bean class and are further described in the deployment descriptor. As a naming convention all select methods are always called ejbSelect<method-name-suffix>

Create methods For each **create method** defined in the home interfaces there must be a matching **ejbCreate method** and a matching **ejbPostCreate method** in the bean class. The return type of the ejbCreate method must be the type of the component interface, but in the implementation return null, the return type of ejbPostCreate must be void. The ejbCreate method must initialize the value of all CMP fields, while the ejbPostCreate must initialize the value of all CMR fields. Furthermore as a naming convention in the bean class the name of ejbCreate methods must always be ejbCreate<method-name-suffix> and of ejbPostCreate methods always ejbPostCreate<method-name-suffix>. In the home interfaces the naming convention for create methods is create<method-name-suffix>.

Home methods Correspond to ordinary static methods. Home methods must be provided with implementation in the bean class and must be specified in the home interface. The naming convention require the name of the home method in the home interface not to start with 'find', 'create' or 'remove', while the name of the corresponding home method in the bean class must be ejbHome<name-of-method-in-home-interface>.

Business methods Correspond to ordinary non-static methods and must have an implementation in the bean class and be specified in the component interface. As a naming convention names of business methods may never start with 'ejb'.

Private methods Are not a method type specified in the EJB2.0 specification, but is presented here as a category for the only other type of methods allowed in Entity Beans. Private methods correspond to ordinary methods declared private. Private methods can only be used be the bean itself. They may not be called by clients, and are thus not exposed in the interfaces. Private methods must have an implementation.

This section has presented an overview of some aspects of an CMP Entity Bean. The specification of EJB2.0 specifies even more details of the construction of an CMP Entity Bean and

the requirements to which it has to conform. The purpose of this section is to illustrate that the development of Entity Beans can be cumbersome. There are rules that are not checked by a compiler, and writing XML deployment descriptors defining relations and methods is not the most thrilling assignment and will often result in many unnecessary mistakes.

1.2 Generative programming and code generation

This section introduces the fundamental concepts and benefits of generative programming and code generators.

Generative programming is the idea of moving from developing one-of-a-kind software systems to the automated manufacture of wide varieties of software systems. *"Generative programming is a software engineering paradigm based on modeling software system families such that given a particular requirements specification a highly customized and optimized intermediate or end product can be automatically manufactured on demand from elementary reusable implementation components by means of configuration knowledge."*[12, p.5]

1.2.1 Generators

In the context of information technology the term generator is very general and covers many different technologies: *"...compilers, preprocessors, metafunctions that generate classes and procedures, code generators (e.g. as in CASE tools), transformational components, and more."*[12, p.334]

The basic tasks of a generator are:

- Check the validity of the input specification and report warnings and errors if necessary.
- Complete the specification using default settings if necessary.
- Perform optimizations.
- Generate the implementation.

Figure 1.3 shows the main tasks performed by a generator. The figure shows how input is given in the form of a system specification, subsequently the generator performs its tasks and generates a system implementation as output.

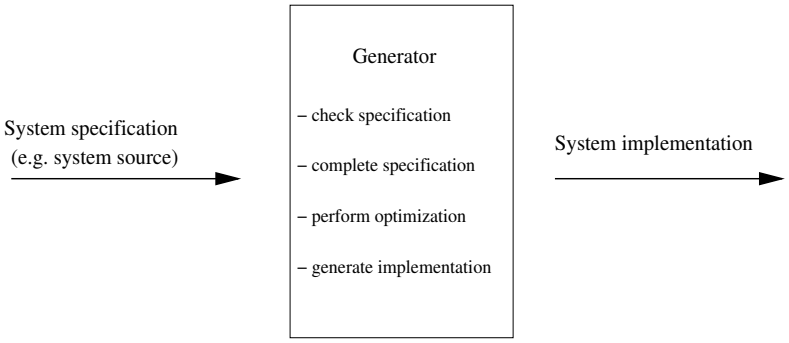

Figure 1.3: The basic tasks of a generator

For some generators it is necessary to parse the input specification to a suitable data representation. The input specification could be some kind of text that is parsed to an abstract syntax tree (AST) used as an internal data representation which the generator will work on. This is the case for most compilers. In other cases the input specification can be used directly.

Generators themselves can constitute a large program, why K. Czarnecki and U. Eisenecker [12] has the following suggestion for designing generators: *"Finally, generators that implements complex specification notations will be quite complex themselves, and we should modularize their internal design. In other words, we will implement larger generators as sets of smaller, cooperating generators."* ([12] p. 335)

1.2.2 Code generators

"A generator is a program that takes a higher-level specification of a piece of software and produces its implementation."([12]p. 333) This is the exact purpose of a code generator. Given some input specification the code generator must produce the necessary source code for implementing the system.

To make code generators a success in the development process it is necessary to perform a thorough domain analysis for the domain of the code generator. A domain is a collection of similar problems or a collection of similar applications. When examining a collection of applications, differences, known as *variabilities*, must be determined. Finding the variabilities also implies finding *commonalities*, aspects that does not vary from application to application.

Commonalities are the decisions or assumptions made during domain analysis about what is common across the domain.

Variabilities are the differences that are identified during domain analysis. These are also referred to as build-time or run-time decisions, since they represent application specific variations.

The variabilities are provided as input to the code generator and commonalities are build into the the code generator. The output is thus a mix of the commonalities (from the generator) and the variabilities (from the input file)[5].

1.2.3 Why develop code generators?

The number one reason for developing a code generator is to enhance development time and increase productivity. The time saved by creating a code generator is dependent on the number of variabilities the generator must handle. If there are too many variabilities not much is gained, or as Craig Cleaveland puts it *"the difference between the specification and the generated code will largely determine the increased productivity obtained by using program generators."* [5, p. 16]. Not much is gained if one has to write the exact same amount of code for the specification as for creating the system from scratch, unless the generator checks correctness, consistency, resource usage or similar issues.

So what is gained by creating a code generator in a domain which covers many different applications but also holds many commonalities:

- Ease of maintenance. Specifications are often easier to understand, modify and debug than the code that implements the specification. Maintaining a system could simply be done by adding changes to the specification, and generate the system once more.

- "Separation of concerns" becomes possible. During development of applications it will be possible to work in separate groups working on different parts of the same application. The generator will assemble the application from the parts.
- Multiple products: Besides generating source code, the generator could be extended to generate documentation, cases or test scripts as well.
- Multiple variants: Program generators can easily create multiple variants of a program.
- Consistency of information: Updating a system can result in inconsistent information. A code generator can ensure consistency and simplify system updates. By updating the specification and regenerating the system, information remains consistent.
- Correctness of generated programs. In large applications with thousands of lines of code, errors easily occur. When generating programs, the code is more reliable than if the same code was written by hand over and over again.
- Multiple binding times. Performance can be improved because program generators provide multiple binding times. Normally when writing programs there are two binding times: Compile-time and run-time. A variable bound at compile-time cannot be changed at run-time. Variable bound at run-time can influence the performance. As an example take a program that during run-time needs some kind of configuration data, which is located in a separate file. By reading the file during run-time the program would have a slow start-up. If data were available at compile-time, they could be incorporated into the source code, the consequences being that adding changes to the data would mean adding changes to the source code which always enhances the risk of errors in the program. But in generative programming there is yet another binding time, the generation-time. By letting the generator read the data file during generation-time, the data would be correctly included in the generated code and the generator could check the consistency and correctness of the data. Both performance and correctness of the program would be improved.

As stated there are a number of good reasons for developing a code generator. Especially if one is working in a domain with many commonalities.

1.3 Applying generative programming to the EJB2.0 domain

We have now introduced two separate fields of computer science. First the field of object oriented distributed systems illustrated through J2EE and EJB2.0 and secondly the field of generative programming. This section serves the purpose of connecting the two subjects presented so far. The point of connection is the idea of developing a code generator for EJB2.0. But according to Graig Cleaveland [5] one must start of by posing the following questions.

Is the domain clearly defined? The domain in focus for this code generator is EJB2.0. The domain is very clear, since it is precisely defined by the EJB2.0 specification.

Will it pay off to develop a generator? As introduced in the section about EJB2.0 and Entity Beans there are many requirements to conform to when developing Entity Beans. By developing a code generator it will be possible to ensure correctness and observance of all requirements in the generated applications.

Maintaining the generated application will become easier since changes are added only in the specification input and not throughout the bean class, interfaces and deployment descriptors.

EJB developers will be spared the monotonous work of writing the same code over and over, such as the 7 callback methods and XML tags for the deployment descriptor.

So there is no doubt that crafting a EJB2.0 generator would pay off.

What are the time perspectives? The question in focus here involves the time perspective of the domain. Is the domain a fast changing domain so that by the time the generator is finished, it will already be obsolete, because the domain has changed? The domain of object oriented distributed systems is developing fast, and at some point in time an EJB3.0 specification will arrive, along with other standards. But it is our belief that developing a EJB generator will not be obsolete, since the standard has already proved its sustainability and is widely used and implemented.

Chapter 2

Method

2.1 An open source project based on open source projects

This master thesis is a software development project in which the code generator tool 3E is developed. 3E is an open source project that complies to the following criteria of the MIT open source license:

1. Free Redistribution
   3E does not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. 3E does not require a royalty or other fee for such sale.

2. Source Code
   3E includes source code and is distributed in source code as well as compiled form.

3. Derived Works
   3E allows modifications and derived works, and allows these to be distributed under the same terms as 3E.

3E is not only an open source tool, but is also integrated, tested and distributed with other open source tools. 3E is build upon open source projects such as Castor XML and the Apache projects Velocity and Ant. 3E implements several features from open source projects such as MiddleGen and TEGA. Finally JBoss, the open source version of the J2EE application server has been used as a testing facility.

2.2 Development environment

In order to create a development environment where every day tasks are performed easily, ensuring backups and managing configuration, we chose to use the build tool Ant [1] and the CVS client Tortoise CVS [26].

Ant is a command line build tool, where various targets are described in a XML build file. An extensive build file for Ant has been created to take care of tasks such as building 3E, building Javadoc, compiling generated code, creating .jar, .war and .ear files needed for deploying J2EE applications, deploying J2EE applications and running 3E.

Tortoise CVS is an open source windows client system that lets you work with files under CVS directly from Windows Explorer. CVS was used for two purposes: To provide record keeping and to facilitate concurrent editing of files.

2.3 Development process

The development process of 3E has followed the prototyping paradigm. [22]. The prototyping development process begins with the developer and customer both identifying requirements for the system. Secondly, a prototype is created which the customer reviews and yet new requirements are added. An iterative process of more and more advanced prototypes will result.

Requirements for 3E were gathered by examining existing codegenerator tools and by interviewing an EJB developer. The identified requirements were prioritized and the first prototype was an implementation of the most basic requirements. An iterative process occurred as more and more requirements were implemented in 3E and feedback was given by the EJB developer. The fourth prototype developed is the final version of 3E presented in this thesis.

2.4 Terminology

To eliminate misinterpretation of terms, this section will define terms used throughout the report:

Application specification: A specification of the variablities in an application to be generated. The application specification for 3E is the XML input file.

Application model: The internal data representation of an EJB application to be generated by 3E.

Entity Bean: The term is used as an abbrivation for 'Container Managed Persistence Entity Enterprise JavaBean'.

EJB application Is an application that contains EJB components and a deployment descriptor. 3E generates an EJB application from an application specification.

J2EE application Is an application that contains an EJB application and client side programs such as Java Server Pages or Servlets.

Chapter 3

Analysis of requirements

The analysis of requirements consists of two parts. Section 3.1 is an overview of several other EJB source code generators. This section will provide us with ideas of good designs and relevant generator technologies. Section 3.2 is a study of functional requirements for a EJB generator tool. The study is carried out as an interview with an EJB developer and provides us with a list of central functional requirements.

3.1 EJB generators

As part of the preliminary analysis we have examined a wide range of similar EJB generators. We have looked into commercial Integrated Development Environments (IDE) and open source projects to find out how we can contribute to and improve the efficiency of the development process for EJB developers. Tools for generating EJBs have been developed before. To develop an improved product we have examined strengths and weaknesses of the existing tools. The main focus of this analysis is not on functionality, but rather on design and technologies. We have selected a few tools to be presented here by the following criteria:

- The tool uses an interesting design or technology
- The tool demonstrates problems we want to avoid

3.1.1 XDoclets

XDoclets [28] uses Javadoc tags and the Javadoc mechanism to generate al sorts of files from a template. XDoclets contains an extension call EJBDoclets. EJBDoclets is an open source project started by Rickard Oberg [8] and is used to facilitate the coding of Enterprise Java Beans. The developer has to code the bean class himself and provide it with the appropriate XDoclets tags. The tags are used to generate the remote and local interfaces and the deployment descriptors. A wide range of Javadoc tags is available, including vendor specific tags. Vendors such as Weblogic and Jboss are represented with tags of their own to generate server specific deployment descriptors.

An advantage of the XDoclets is the possibility to make your own tags and for J2EE aplli-cation server vendors to make vendor specific tags. A disadvantage of XDoclets is the fact that Javadoc tags are written within the bean class. First of all the developer has to write his own bean class, not receiving any help to specify or check get and set methods, remembering which methods are returning what or which methods are to be defined. Secondly, the bean class will

become vendor specific according to the vendor specific Javadoc tags. Thus reuse of the bean class becomes harder.

3.1.2 MiddleGen

MiddleGen [17] is an open source tool to generate EJBs. The basic technologies this tool is build upon are Java, JDBC and XDoclets.

The tool is GUI based and by accessing a specified database it can retrieve enough information to model the corresponding EJBs. From this information the abstract bean class is generated and provided with XDoclet tags, and the XDoclet technology is then used to generate the remaining interfaces and deployment descriptors.

On the surface the application seems to be able to do many of the things we want our tool to do. Upon taking a closer look at the generator code of the application we found that the code is generated by appending Strings to a PrintWriter object. Thus the generator code is full of out.println statements and there is no seperation between generator code and code to be generated. As a consequence the generator code of the MiddleGen tool is not easily maintainable and does not encourage reuse. E.g. from middlegen.codegen.BeanWriter.java [17]:

```
private void writeClassFooter(PrintWriter out) {
   // EntityBean interface
   out.println("   public void setEntityContext(javax.ejb.EntityContext entityContext) {");
   out.println("      _entityContext = entityContext;");
   out.println("   }");
   out.println();
   out.println("   public void unsetEntityContext() {");
   out.println("      _entityContext = null;");
   out.println("   }");
   out.println();
   out.println("   public void ejbLoad() {");
   out.println("   }");
   out.println();
   ...
}
```

A second disadvantage of the MiddleGen application is the fact that it relies on the XDoclet technology and thereby inherits the disadvantages of XDoclets, such as a vendor specific bean class.

3.1.3 EJBBuilder

EJBBuilder [7] is an open source plugin for the Eclipse platform [6]. Though not completed it is the vision to provide an Enterprise JavaBean wizard with support of Entity, Session and Message-Driven Beans. All source files necessary for each type of EJB will be generated by the tool.

The generator technology used in EJBBuilder is the Apache project Velocity[25] – a template technology very similar to Java Server Pages. The Velocity templates used to generate EJB source code are very easy to read. We found that the force of the EJBBuilder was its templates and the ease of modifying and correcting these.

On the other hand we could not learn much from their architectural design since the tool was build as a plugin for a much larger application, and as such the class controlling the source code generation was very hard to understand.

3.1.4 EJBGen

EJBGen 2.0 [9] is an EJB extension module for the SrcGen project [24]. It generates Java and XML source files based on meta-data supplied in the form of an XML document that conforms to a DTD/XML Schema describing the structure of an EJB. EJBGen is intended to be a flexible source code generation library for EJBs. The reason for making it a library rather than an application is that a library is more robust – you don't have to worry about details like a user interface. This makes it possible to use the tool with other applications like for instance Ant.

The tool uses DTDs to model the core attributes of the EJB. It specifies the data needed by the Java objects in the library which is used by the framework to generate the EJB source code. The generator uses XML templates to generate the bean classes and interfaces.

We found the idea to use XML and XML schema as input interesting and the idea of integrating the tool with other applications such as Ant very compelling.

3.1.5 JDeveloper

JDevelopler is an IDE from Oracle which supports the complete development life-cycle of EJBs.

Options and wizards in JDeveloper seem endless. Anything can be specified from a GUI where help is provided by wizards or UML modelling. Deployment descriptors and JSP pages can be generated, packing of jar, war and ear files can be done as well, and even deployment is possible from the IDE. But yet such a tool has several drawbacks:

- It is very expensive ($1000 pr. license)
- It only supports development of EJB1.1 (at the moment)
- It only integrates with two vendors (Oracle and Bea)
- Access to source code is not provided, making extensions, adjustments and customization of the program impossible
- It does not integrate with other development tools

JDeveloper from Oracle is a program where options and wizards seem endless. We can learn a great deal from the product by observing what to specify and how to specify certain variabilities.

3.1.6 TEGA – The EJB Generator Application

TEGA is another open source EJB generator developed by Lund & Bendsen. TEGA provides an API for expressing Entity Beans. In an earlier project we have examined TEGA and its design thoroughly and discovered several drawbacks, most of all due to the fact that TEGA is not completed. First of all the tool does not conform to the EJB2.0 specification since many features are not expressible in TEGA. Secondly the generator part of the source code to TEGA has the same drawback as MiddleGen: The generator source code and the code to be generated are mixed together.

Despite the drawbacks, TEGA has provided us with an example of how to design a generator tool, see figure 3.1. The main thought behind TEGA, though not fully implemented, is to parse some kind of input to an internal data representation, and from here generate the corresponding source code for EJBs, thus obtaining a nice seperation of input, internal data representation and generation of source code.

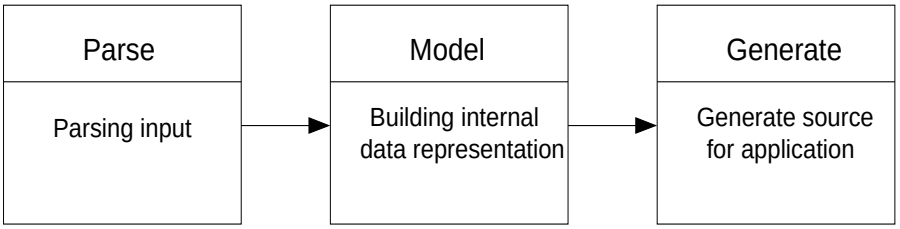

Figure 3.1: The design of TEGA

3.1.7 Requirements

We have presented a few of the tools we have examined, and described their forces and drawbacks. Other tools such as Jeewiz, UML2EJB, Ejen and Jenerator, have been examined as well, but are not presented here. From these analyses' we have seen examples on what to do, and what not to do. Following is a list of requirements for the 3E tool extracted on basis of these analyses'.

- 3E should be an open source project and freely available.
- 3E must be easy to maintain and extend with improved functionality
- 3E must not be vendor specific or generate vendor specific source code
- 3E must provide good documentation and user manuals
- 3E should be able to integrate with other development tools

The requirements are the result of the analysis of several code generating tools. None of the tools tested seems to be able to meet all of these requirements. The requirements are not specific functionality requirements but rather general concerns on how to improve an EJB development tool. These requirements will become our guidelines for designing and implementing 3E.

3.2 Requirements analysis

In order to provide us with a list of features and functional requirements for an EJB generator tool, and to give this thesis a touch of real life, we found it important to talk to people using J2EE. Thus we made arrangements to interview Morten Madsen, a software developer employed at TietoEnator, a Danish software engineering company.

3.2.1 Interview with an EJB developer

Morten Madsen is currently working on a project where J2EE technology and EJB development is the main substance. The role of Morten Madsen has been to evaluate the technology and find appropriate development tools to aid the development of EJBs. TietoEnator has formed a partnership with Oracle, using their application server, and is considering to use Oracle IDE JDeveloper.

The main reasons for choosing JDeveloper is that it is easy to use and develop EJBs with. Another important factor is that JDeveloper is compatible with the Oracle application server. In

Morten Madsens search for an IDE JDeveloper was the only one which could generate deployment descriptors for the Oracle Application server.

The only concern when using Oracle products is a fear of becoming vendor dependent. The fact that all EJBs are generated by an Oracle tool to run on a Oracle server not knowing what happens beneath the surface, might create problems. As Morten Madsen states, they have no guarantee that Oracle didn't implement other services than stated in the J2EE and EJB specifications, and when generating the EJBs, these might become vendor specific. Thus making a shift to another application server could become a challenge.

While talking about developing an EJB generator tool, Morten Madsen has the following suggestions for requirements:

First of all, and the most important one, he states, is that the tool should be easy to use. Everyday tasks should be implemented in such a way that the user should not open a user manual, but just start working. Simple things should be kept simple, while on the other hand, for special features not often used, in-depth knowledge of the tool is necessary.

Furthermore the interview produced the following list of requirements to the system. The list contains both very specific requirements, domain limits and some design consideration.

What to generate The tool should be able to generate CMP Entity Beans, deployment descriptors and only the local interfaces. There is no reason to generate remote interfaces since a Session Bean will always be used as a facade. The need for generating Session Beans and Message-Driven Beans was not highly prioritized. As Morten Madsen states it: *"Session and Message-Driven Beans have too many variabilities, so not much is gained by creating a generator for these types of EJBs. I would rather write these myself."*

JSP pages Test JSP pages may be generated but was not really considered important.

Value objects Value objects might be generated but was not highly prioritized. Value objects for the Entity Beans might be generated, but value objects or transfer objects between the application and client programs, should not be generated.

Relations The ability to express relations between Entity Beans.

Transaction The ability to express the transaction level.

Error Messages It is important to get error messages that point back to the input file, not to the generated code.

Database access The tool should be able to retrieve information about existing database tables and columns in order to generate EJBs specific to a database already running. Not having to look up database information will save development time and avoid the developer mistyping table and column names.

Primary keys The tool should accomodate primary keys of any type. It would be useful for the tool to support automatic generation of primary keys, especially GUID (globale unique identifiers).

Compatibility It is important that the system can be extended to improve compatibility with e.g. various application servers. It would be an advantage for Morten Madsen, if he could easily extend the generator to generate the vendor specific Oracle deployment descriptor.

The list above states the most important requirements that an EJB generator tool should comply with according to an EJB developer. The most obvious one though is not stated. The generator tool should conform to the EJB2.0 specification.

The list of requirements from Section 3.1.7 page 20, the list above and the EJB2.0 specification are thus used throughout the design and development process of 3E.

Chapter 4

Design

In this chapter we will discuss how to implement the EJB generator tool. In previous chapters numerous requirements have been listed. Now we need to decide how to put things together to meet those requirements. As yet only two things are certain with regard to the design:

- The input to the generator should be an application specification.
- The output from the generator should be the required Java files and deployment descriptors to implement the specified EJB application.

In between the following four tasks need to be solved:

- The application specification needs to be validated. Is the specification syntactically well-formed? Does the specification contain all the information required by the generator?
- All available information about the application must be retrieved from the application specification. We only want the specification to contain the absolute minimum information required to generate the EJB application. Information that can be inferred from other information in the specification should be optional or not present.
- The collected information needs to be processed to obtain all required information to build the EJB application. The generator must in some way be able to examine the collected information and add new information, which may either be inferred from the application specification or be default values and commonalities dictated by the EJB2.0 specification.
- From the obtained information the code implementing the EJB application must be generated.

4.1 Program structure

In the previous chapters we argued that the generator should be easy to understand and easy to extend for other developers. To ensure this demand a good program structure is important. A problem will often be easier to understand and to solve if you can split it up into smaller separate problems, each of which can be solved independently.

In Section 1.2.1 on page 11 we described how a generator would often get the job done by splitting it up in three different parts: 1) parse an input to some internal representation, 2) do some extra work on this and 3) generate an output. E.g. this strategy is reflected in the structure

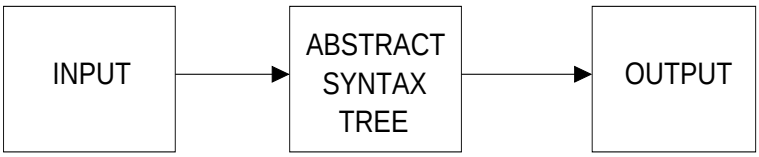

Figure 4.1: General structure of a generator

of TEGA as shown in Figure 3.1 on page 20. Looking at this structure from a higher perspective, what you have is an input, an abstract syntax tree (AST) and an output, see Figure 4.1.

Concerning the structure of our generator, this seems an obvious choice as well. We shall refer to the input as the *application specification*, the abstract syntax tree we shall call the *application model* and the output will be an *EJB application*. In the previous section we mentioned four different tasks that need to be solved by the generator. These tasks actually infer additional structure to the generator by making four seperate parts, each dealing with one of the tasks mentioned:

1. One part dealing with how to validate an application specification. This part would be working on the application specification alone.

2. One part dealing with how to collect information from the specification. This part would be the bridge between the application specification and the application model.

3. One part dealing with how to check consistency and extend the collected information. This part would be working on the application model alone.

4. One part dealing with how to generate code from the final set of informations. This part would be the bridge between the application model and the generated EJB application.

Hence we end up with a structure as the one shown in Figure 4.2.

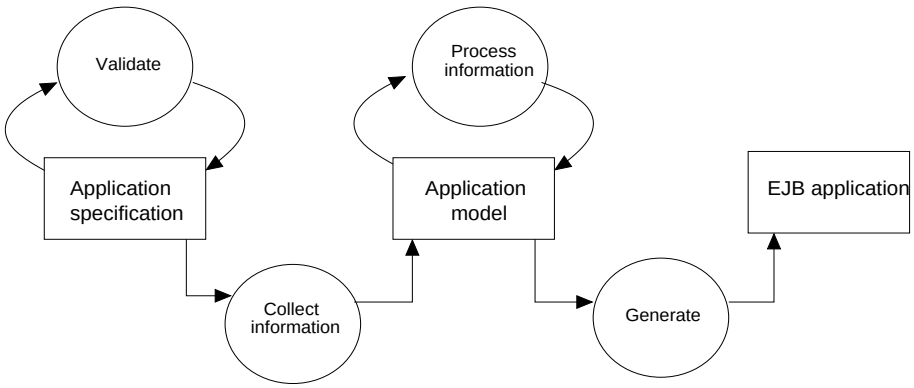

Figure 4.2: Structure of our generator

This structure leaves us with a collection of smaller, separate concerns. In the following sections we shall discuss each of them separately.

4.2 The application specification

The input can be given in many ways, but the main point is to reflect an application specification.

In most of the existing tools examined in Chapter 3 the user specifies the application by interacting with the generator through a GUI. One exception to this is TEGA, in which the application specification is made as a Java object model directly by the user using the TEGA API. Another is EJBGen, which lets the user specify the application in a XML-document.

What are the requirements on the input? Firstly it should be *easy to understand* and *easy to edit*. The input should therefore be in a format well-known to or easily learned by most developers. Secondly it should be possible in some way to *validate* the input. The generator must be able to determine whether the input is valid according to what the generator expects, and if not to reject the input as invalid. Thirdly the input should be *well-structured*. This will most likely improve the developer's understanding of the input, and will also make it easier to collect information from it. As a fourth point the input should be *easy to save*. The developer might not finish the input all at once and should therefore be able to save it for later. When code is generated based on the input, the developer should also be able to review and edit the input repeatedly. Finally it should be considered how suitable the input format is for *defining Java code*. An application specification will usually also contain definitions of methods written in Java. As Java syntax contains many special characters, it should be considered how easily these can be escaped (if necessary).

4.2.1 Selecting an input format

Inspired by the existing tools and by our own experience concerning interacting with a program, the input could be given by using 1) a GUI, 2) an API to build an object model representing the application or 3) XML. Note that the object model would be implemented in Java since this is the programming language we know best. Although a GUI might be the most obvious choice with regard to the usability of the generator, we will at this point exclude this option from any further considerations, for the reason that we do not want to spend a lot of time developing a good GUI. Instead we will leave this option as an opportunity for later improvement of the generator. This leaves us with two other options: Using an API to build an object model, or using XML. How does each of these two options fulfill the requirements pointed out above? The answer is given in table 4.1 on page 26.

The choice between an object model and XML is not straightforward. It would be fair to assume that most Enterprise JavaBean developers are familiar with Java which speaks in favour of choosing the object model input and Java as the object modelling language. Then again, most of those developers would also be familiar with XML as the deployment descriptors for the EJBs are written in XML.

Our choice will be XML for two reasons: 1) The program should not imply good knowledge of an object modelling language such as Java and 2) Using XML makes the input independent of the rest of the program. This means that no obligations are made to a specific programming language for implementing the generator, and hereby the re-usability is improved.

4.2.2 A language for the application specification

We still need to define *how* the application specification should be expressed using XML. For this purpose we examine XML Schemas more closely.

	Java Object model	XML
Easy to understand	Easy for users familiar with object modeling languages, i.e.Java developers	A common framework for structuring and exchanging information. Rather self-explanatory and therefore fairly easy to understand, even for an inexperienced user
Easy to edit	True for users familiar with Java	Simple and easy to learn, but requires a lot of writing. Various XML editors can be found to ease this task
Can be validated	A Java compiler is able to check that the structure is legal, but cannot test the content of objects. This could be done by implementing additional traversals of the object model, e.g. by using a visitor pattern	The structure and to some extend the content could be validated using a XML Schema (XML Schemas are made for validating XML documents)
Well-structured	A Java object model is by itself a tree structure. For the user the structure of the object model is not necessarily obvious	XML is simply made for structuring information and will by itself reveal this structure quite clearly, depending on the editor
Easy to save	By saving the Java file in which the object model is build. This will also be the file from which the object model can be edited	XML is simply saved as an XML file
Suitable for defining Java code	Can be defined in a String object using Java's own escape features	Can be contained in a string element using the CDATA section or the XML escape features to escape special characters

Table 4.1: Object model versus XML

XML Schemas

This section assumes that the reader has a basic understanding of XML and how an XML document consists of elements and attributes.

An XML Schema is itself written in XML and describes the *structure* of an XML document. The purpose of an XML Schema is to define the legal building blocks of an XML document. It does so by ([27])

- defining elements that can appear in a document
- defining attributes that can appear in a document
- defining which elements are child elements

- defining the order of child elements
- defining the number of child elements
- defining whether an element is empty, can or must include text
- defining data types for elements and attributes
- defining default and fixed values for elements and attributes

In other words XML Schemas gives us a way to define the syntax of our own XML-based language. By letting an XML document refer to an XML Schema, the author claims that the document is intended to be valid with respect to the schema (not that it necessarily is valid). Being valid means that the XML document fulfills the requirements defined by the XML Schema: It contains only legal elements/attributes in the right number and order, the elements/attributes contain values where required and values are of the defined type. If this is not true for the XML document, a schema processor will reject it as being invalid [23]. Thus an XML Schema does not just give us a way of describing our own language for an application specification in XML, it is also useful for the task of validating the input.

XML Schemas is only one of many schema languages. Other options exist, e.g. DTD (Document Type Definition). Currently it seems that XML Schemas are the most powerful choice, so no further description of the alternatives will be given in this project [27, 23].

4.3 Validating the application specification

The XML Schema can to some point validate the XML input file. Exactly how much can be validated depends on what is defined in the schema. As the very minimum the XML Schema should check that the XML input file actually contains an application specification. This means that the XML Schema at least should define names and types of elements and attributes necessary to describe an application specification. By doing so the schema can validate whether all elements and their attributes in the XML input file are part of the application specification language, the language described by the XML Schema itself.

As suggested in Section 4.2.2 about XML Schema, it can also define and thereby validate many other things. E.g how many times a certain kind of element can appear in the document (using attributes such as minOccur/maxOccur). Or the order in which some elements should appear (sequence/choice tags). It can also make sure that a certain attribute always has one of two specified values (using the restriction tag), specify whether an attribute is optional or required (use='optional' or 'required') and so on [27]. These features should be used to the extent that it helps the developer to make a meaningful XML input file — meaningful implying that, for instance, if an Entity Bean is described, one should also define a name for this Entity Bean and so on.

4.4 Collect information from application specification

In this section the following question needs to be answered: How can information from the application specification become available to the generator? Since the input will be given as an XML document, the answer to this question is straightforward: By using an XML Parser.

4.4.1 XML Parsers

To use data from an XML document in a program, the data in the XML document must be read and passed to the program. This is what an XML Parser does. First it separates XML tokens from data to make sure that the XML document is *well-formed*, and then it passes the data to the client application that invoked the parser. If the parser detects a violation of XML rules, it reports an error and stops parsing [11, 14].

Lots of XML parsers are available. Most of them support one of two popular parsing standards: The SAX API or the DOM API.

DOM — Document Object Model

In the DOM world an XML document is viewed as a tree-like data structure. Like the name DOM implies, parsers that implement DOM will create an object model in memory that represents the content of the XML document. The object model reflects the hierarchical data format of XML, where elements can be nested in other elements and attributes can be attached to elements. The object model is available and can be modified by the client application, but only after the entire XML document has been parsed [11, 14].

SAX — Simple API for XML

SAX is an event-driven way of parsing XML documents. In the SAX world an XML document is not viewed as a data structure, but as a stream of events generated by the parser. Parsers that implement SAX generate an event each time an XML construct is recognized. By listening to these events the client application can either save or process data as they are passed by the XML parser. The client does not have to wait for the entire XML document to be parsed before the data from the XML document becomes available. Parsers that implement SAX do not store data from the XML document during parsing, and for this reason SAX has no support for modifying or writing XML document data [11, 14].

4.4.2 Some options

In the following section we will discuss three different options. An important issue for all three of them is how the data from the XML document will or can be represented after parsing. As suggested earlier, data from the input should at some point be represented in an application model for further processing by the generator. For this reason the best solution concerning parsing would be if the application model was created during the parsing process and ready for further modification right after the parsing had finished. Also, we would like to have some control over how the application model is structured.

Option 1: Implement a DOM-parser

As the first option we could implement our own XML Parser using the DOM API. The advantage of DOM in our project is the creation of an in-memory object model available for modification. This could be our application model. In other projects the reasons for not using DOM are the efficiency problems suggested earlier: The object model will not be available until the entire XML document is parsed and as the entire document ends up being represented in memory, implementations of DOM often become very memory consuming [11, 14]. These concerns might not be a big issue in this project since high performance of the generator is not our top priority.

For us the most vital concern using DOM would be the lack of control with regards to the creation and structure of the in-memory object model, which should then also be our application model.

Option 2: Implement a SAX-parser

A second option would be to implement an XML Parser using the SAX API. This would give us full control with regard to the representation of how the data from the XML document would be represented in memory. Suitable classes for representing data should be implemented. Say the XML document contained an XML element <entity-ejb> to represent an Entity Bean. Then a class to represent Entity Beans should be implemented and all data contained in an entity-ejb element should be passed to an object of this class. All classes implemented to represent data from the XML document would naturally become the building blocks of our application model. The drawback of using SAX is the amount of work it would require, not only implementing the classes for the application model (which we would like to do), but also to implement all the parsing code. This would be acceptable if we only had to do it once, but as the developing process is iterative, we would most likely have to modify the parser repeatedly.

Option 3: Use an existing parser

A third option exists: Use an existing XML parser. This would save us the time of doing everything ourselves. Not surprisingly this option does also have drawbacks: Again it seems that we will lose some control. The main concern is once more the internal representation of the data or the application model. Fortunately it turns out that tools for XML parsing exist that allow the user to employ an existing XML Parser and still define his own internal representation. One such tool is Castor XML.

4.4.3 Castor XML

The Castor Project is an open source data binding framework for Java [3]. Castor XML is one feature of the project dealing with mapping between a Java object model and XML. Note that it can map both ways. It is also worth noting that it uses Java as the object model language which would also be our choice of programming language for implementing the application model. Using Castor XML gives us two options for handling the parsing process: 1) let Castor XML generate the Java source code for the object model from an XML Schema, or 2) make Castor XML map data between XML and our own Java object model using a so-called *mapping file*.

The first option was tried and found un-satisfying. Castor XML's way to implement the required classes does not conform to how we would like them to be. Also, this solution puts another perspective on the XML Schema, not to mention that it would tie the schema and the classes for the object model very tight together: If anything changed in the schema, the classes would have to be regenerated by Castor and also, the classes could only represent data reflected in the XML Schema.

The second option gives us the opportunity to implement the application model ourselves while Castor XML does all the work with regard to the parsing process. The link between the XML input file (and thereby the XML Schema) and the application model will be the *mapping file*, and only this has to be modified if something changes at either end. The next section will discuss more closely why implementing our own application model is such a desirable goal in this project. For now we just conclude that Castor XML and the mapping file solution seems to be a good choice. Of course we will still have to write the mapping file, but that appears to be a

moderate task. Example: Figure 4.3 shows an XML document (order.xml) containing data about an order.

```
<Order reference="12343-AHSHE-314159">
   <Client>
      <Name>Jean Smith</Name>
   </Client>

   <Item reference="RF-0001">
      <Quantity>10</Quantity>
      <UnitPrice>8.95</UnitPrice>
   </Item>
</Order>
```

Figure 4.3: An XML document (order.xml) containing data about an order

Assume that the classes shown in Figure 4.4 are used to make the internal representation when parsing the XML document from Figure 4.3.

```
public class MyOrder {
  private String _ref;
  private ClientData _client;
  private Vector _items;

  public void setReference(String ref) {...}
  public String getReference() {...}

  public void setClientData(ClientData client) {...}
  public ClientData getClientData() {...}

  public void setItemsList(Vector items) {...}
  public Vector getItemsList() {...}
}

public class ClientData {
  private String _name;

  public void setName(String name) {...}
  public String getName() {...}
}

public class Item {
  public String _reference;
  public int    _quantity;
  public float  _unitPrice;
}
```

Figure 4.4: Java classes used to make the internal representation of an order

The mapping file (mapping.xml) should then look as shown in Figure 4.5.

```
<?xml version="1.0"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Object Mapping DTD Version 1.0//EN"
                         "http://castor.exolab.org/mapping.dtd">
<mapping>
   <class name="MyOrder">
      <map-to xml="Order"/>
      <field name="Reference" type="java.lang.String">
         <bind-xml name="reference" node="attribute"/>
      </field>
      <field name="ClientData" type="ClientData">
         <bind-xml name="Client"/>
      </field>
      <field name="ItemsList" type="Item" collection="vector">
         <bind-xml name="Item"/>
      </field>
   </class>
   <class name="ClientData">
      <field name="Name" type="java.lang.String">
         <bind-xml name="Name" node="element"/>
      </field>
   </class>
   <class name="Item">
      <field name="_reference" type="java.lang.String" direct="true">
         <bind-xml name="reference" node="attribute"/>
      </field>
      <field name="_quantity" type="integer" direct="true">
         <bind-xml name="Quantity" node="element"/>
      </field>
      <field name="_unitPrice" type="float" direct="true">
         <bind-xml name="UnitPrice" node="element"/>
      </field>
   </class>
</mapping>
```

Figure 4.5: The mapping file

The example suggests many features about the mapping file, e.g. how to map objects contained in other objects, how to map data into a collection and how Castor XML distinguishes between public and private fields (the need for get and set methods) [4].

The code snippet in Figure 4.6 shows an example of how to use Castor XML Mapping. Note the use of the term *unmarshall* to describe the process of parsing from XML to a Java object model. The term *marshall* is used to describe the opposite process of writing a Java object model in XML format.

It seems we have found a good tool for parsing data from an XML input to our own application model: We do not have to write the parsing code ourselves, and still we have full control over the implementation of the application model.

```
public class main {
  public static void main(String args[]) {
    Mapping mapping = new Mapping();
    try {
      // 1. Load the mapping information from the file
      mapping.loadMapping( "mapping.xml" );

      // 2. Unmarshal the data
      Unmarshaller unmar = new Unmarshaller(mapping);
      MyOrder order =
(MyOrder)unmar.unmarshal(new InputSource(new FileReader("order.xml")));

      // 3. Do some processing on the data
      ClientData client = order.getClientData();
      client.setName("Chris Hanson");
      System.out.println("New client name is " + client.getName());

      // 4. marshal the data with the new client name back and print
      // the XML in the console
      Marshaller marshaller=new Marshaller(new OutputStreamWriter(System.out));
      marshaller.setMapping(mapping);
      marshaller.marshal(order);

    } catch (Exception e) {
      System.out.println(e);
      return;
    }
  }
}
```

Figure 4.6: Example of how to use Castor XML Mapping

4.5 The application model

It has already been implied that the XML input (the application specification) should be parsed to an application model. The application model is the internal representation of the data from the XML input. It has also been implied that the generator will need to do some additional work on the application model, e.g. edit contained information and add new information, and thus we would like some control over the structure of the application model. In the previous section about XML parsers an argument against using a DOM parser was exactly that this would give us less control over the structure of the application model. The best way to obtain control over the structure of the application model is to implement the API to represent the application model ourselves. Two other things would be gained by this solution: 1) making sure that the implementation of the application model is kept simple and understandable and 2) the application specification could then be made by using this API directly instead of using XML; an extra and valuable feature for developers more inclined to use Java than XML.

When generating the EJB application from the application model, the generator will have to look up information in the application model several times. Therefore, the application model must be able to support references in all directions, not only from parent to child, but also from child to parent, and furthermore all information in the application model must be consistent. Note that some children can have more than one parent: E.g. a CMR field which belongs both to an

Entity Bean and to a Relation. By making a complete cross-referencing application model, the generator can look up information in the application model without having to traverse the whole model each time.

The TEGA API for building an object model representing the application (an object model is the input to TEGA) is a good source of inspiration for implementation of our application model. In the TEGA API classes are found to represent many of the elements that are needed in an Entity Bean. Most of the elements are the same elements that can be found in a deployment descriptor. E.g. a class representing an Entity Bean, another representing CMP fields, yet another representing CMR fields, one class for each kind of method and so on. As the TEGA API turns out to work well [19], it was reused in this project as a starting point. Along the way modifications were needed, in some places more heavily than others. Figure 4.7 on page reflects what our application model looks like.

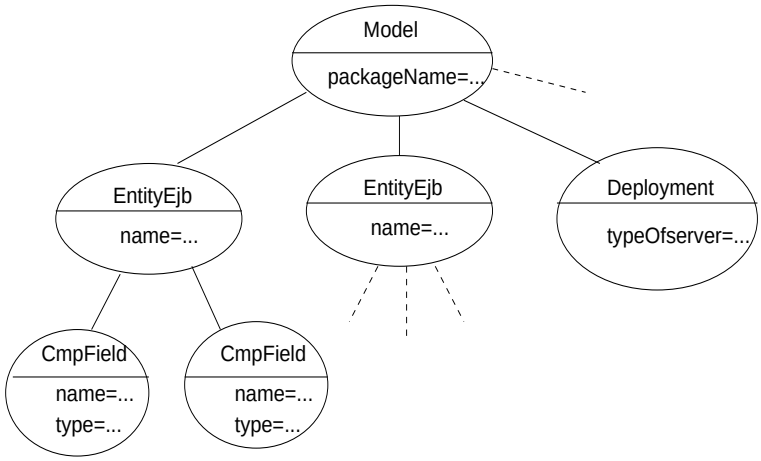

Figure 4.7: Part of the application model in 3E

4.6 How to process information in the application model

After an application model representing an EJB application has been built, its consistency must be checked, additional references must be set and the data must be validated against the EJB2.0 specification.

Right after the application model is built, it will contain the following:

- The minimum information required to generate a meaningful EJB application. This is information about at least one Entity Bean and some deployment information. The minimum information required about an Entity Bean is its name and a primary key field. Additional information such as name and type of CMP fields and descriptions of methods may also be present. Furthermore there may be information about relations.

- References from parent objects to child objects are set — references from child to parent will be missing.

- Default values are present unless overwritten by information from the XML input.

To generate a legal implementation of the defined EJB application with regard to the EJB2.0 specification the following are still needed to complete the application model:

- It should be checked that all data in the application model so far are legal. Here, legal means that no null values or empty Strings are found where information is required (this will to some extent be ensured by the XML Schema, unless the application model API is used to make the application specification). Also the EJB2.0 specification dictates certain rules for naming, e.g. the name of a find method must always start with 'find'.
- As the application model will be traversed repeatedly during code generation it should be possible to go from a child object to its parent object(s). Therefore references from children to parents must be set.
- The EJB specification specifies many rules for Entity Beans, their fields, methods and relations. It should be checked that these rules are followed by the EJB application defined by the application model.

So once again, what we need is a way to traverse the application model and at the same time examine, edit and add information to the objects in the tree.

4.6.1 The Visitor Pattern

One way to traverse an object structure and at the same time examine and modify the objects is to use the Visitor Pattern [10, p.331]. Figure 4.8 shows how to implement a Visitor.

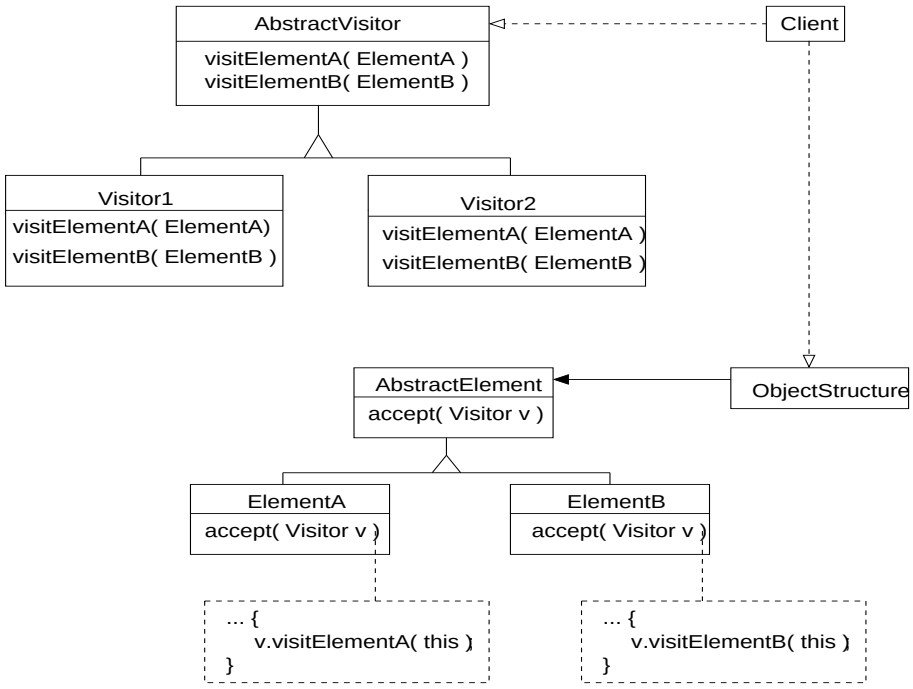

Figure 4.8: How to implement Visitor pattern [10, p.334]

As pointed out above, several different operations need to be executed on the application model. A visitor should be implemented for each operation: One visitor to do data checking, one

to set parent-references, one to check that data is legal with regard to the EJB2.0 specification, etc. The visitors might have to traverse the application model in a certain order, e.g. first the data-check visitor runs through the application model, then the set-parent visitor, then the legal-data visitor and so forth. This is because a visitor may depend on operations done by another visitor. To traverse the application model several times with different visitors is no different from what happens when the Java-compiler compiles a Java source program. As each visitor only visits each object in the application model once, the execution-time will still be linear in the number of objects in the application model. Thus the additional traversals done by the visitors have no severe impact on the performance of the program.

One issue still remains: How to make each visitor traverse all the way through the application model. The description of the Visitor Pattern suggests three options [10]:

1. The object structure itself is responsible for iteration. Each object should from its own accept method traverse all the object's children and call the accept method on each of them.

2. A single iterator is responsible for running through the object structure. This will work only if all objects in the object structure are of the same type.

3. The visitor are responsible for traversing the object structure. Then the traversal code is placed in the visitor. This could mean that the same traversal code would be duplicated in several visitors.

Option 2 can be dismissed without any further consideration. Option 1 was rejected at first sight, the argument being that it did not always seem obvious who was the parent of a child. For instance, does a CMR field belong to the Entity Bean or to the Relation? For this reason the third option was chosen. The drawback of having traversal code duplicated in several visitors is avoided by implementing the traversal in an abstract super class to all the visitors. This means that each visitor should only implement a visit method for those objects that this visitor should actually do something to. The traversal is obtained by calling the visit method from the super class as the last statement in each visit method. At a second look it turns out that the traversal code does exactly the same thing suggested by option 1: Each object traverses its own children and call the accept method on each of them and this without regard to the fact, that some children have more than one parent. Option 1 would therefore also work. The only thing possibly gained by our choice is that all traversal code is gathered in one class: the abstract super class of the visitors. Also, if the traversal order of the application model would have to vary with the kind of visitor visiting, then the traversal code should be in the visitors and not in classes of the application model [10].

4.7 Generating Java and XML from the application model

From the application model containing all necessary information about a certain EJB application the next step is to generate the code implementing the EJB classes etc.

The examination of existing tools and our own experiment in an earlier project [19] suggests three ways to do it. One way is to write code in Java to create the required files and write text to these files, so that in the end the created files implements the application. This is done in TEGA. The drawback of this method is that the generating code tends to be messy, creating several String objects and a lot of write-operations mixing generator code and code to be generated [19]. Another way is to use Java Server Pages (JSP) and Tag Libraries. This turns out to be a good

option concerning Java code generation [19]. The question is how well suited it is for generating XML (the deployment descriptor). The third option is to use Velocity as in EJBBuilder.

An important goal in this project is to make everything well-structured and understandable in order to ease the task of other developers, who may wish to extend the generator. For this reason the first option is rejected. The two remaining options are discussed more closely in the following sections.

4.7.1 JSP and Tag Libraries

JSP is originally made to generate HTML pages to be shown in a browser, but nothing prevents one from using JSP to generate something else. Commonly JSP executes on a server, but it will presumably be possible to separate the JSP generator from the server.

In an earlier project we showed that JSP and Tag Libraries could be a good way for generating Java code based on information in an application model [19]. A Tag Library was defined implementing tags to traverse the application model and gather information needed in the JSP file to generate the Java code. An example of generating Java code from JSP can be found in appendix A on page 90.

But how suitable is JSP and Tag Libraries for generating XML? At first it seems that there might be a conflict as both Tag Libraries and XML uses tag elements (< >). We found two different ways to avoid this conflict.

One way is to use Java code instead of Tag Libraries to gather information, as e.g in Figure 4.9. The example gathers information from an XML document parsed to a list of elements and not an application model, but still it shows the main idea of using Java code instead of Tag Libraries. Figure 4.10 shows the generated XML.

```
...

<?xml version="1.0" encoding="UTF-8"?>
<class>
<% while(ir.hasNext()) {
   MyElement element = (MyElement) ir.next();
   String tag = element.getQname();
   if(tag.equals("field")) { %>
   <field>
   <% element = (MyElement)ir.next(); %>
     <type><%= element.getValue() %></type>
   <% element = (MyElement)ir.next(); %>
     <name><%= element.getValue() %></name>
   </field>
   <% } %>
<% } %>
</class>
```

Figure 4.9: JSP using Java code to generate XML

The drawback of this option is the readability of the JSP file. Java syntax inside XML syntax does not appear to be a good combination, as the mix of two different syntaxes makes it difficult to see what is generated when reading the code.

```
<?xml version="1.0" encoding="UTF-8" ?>
<class>
   <field>
      <type>java.lang.String</type>
      <name>firstName</name>
   </field>
   <field>
      <type>java.lang.String</type>
      <name>lastName</name>
   </field>
</class>
```

Figure 4.10: XML generated from JSP in figure 4.9

The other way to generate XML from JSP still uses Tag Libraries. An example of using JSP and Tag Libraries to generate XML can be seen in Figure 4.11.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jstl/xml" %>

...

<?xml version="1.0" encoding="UTF-8"?>
<books>
   <book> <x:out select="$xml/books/book[1]"/> </book>
   <book> <x:out select="$xml/books/book[2]"/> </book>
   <book> <x:out select="$xml/books/book[3]"/> </book>
</books>
```

Figure 4.11: JSP using Tag Libraries to generate XML. The XML generated by this JSP file can be seen in Figure A.3 on page 91

The example works on an XML input directly using existing tags for working on XML. The tags used for traversing the XML corresponds to XPath expressions. In 3E the tags should work on the application model implemented in Java and not on XML. Presumably tags implemented to traverse the application model generating Java code could be reused. Additional tags would most likely be needed.

4.7.2 Velocity

Velocity is part of the Apache Jakarta Project [25]. It is a Java-based template engine, meaning you can write Velocity templates, from which all kinds of output can be generated: web pages, SQL, PostScript etc. In this manner Velocity can be used to generate source code, reports and so on, and thereby provides an alternative to JSP [25].

The Velocity template shown in Figure 4.12 can generate the same Java code as shown in Figure A.2 on page 91. In a Velocity template all you need to traverse the application model is a reference to the root element. From the template it is possible to reference methods defined in

```
public class $class.getName(){

#set ($field = $class.getField())
   private $field.getType() $field.getName();

   public $class.getName()($field.getType() $field.getName()){
      set$field.getName()($field.getName());
   }

   public void set$field.getName()($field.getType() $field.getName()){
      this.$field.getName() = $field.getName();
   }

   public $field.getType() get$field.getName()(){
      return this.$field.getName();
   }
}
```

Figure 4.12: Velocity template for generating a Java class

Java code, which means the Velocity template can use methods already defined for the objects in the application model. For instance, the template in Figure 4.12 has a reference to an object named `class`, and this object has methods called `getName()` and `getField()`, which are used by the template to get the missing information. Velocity also has support for features such as e.g. loops and conditional statements.

Generating XML using Velocity is easy, e.g. in Figure 4.13.

```
<?xml version="1.0" encoding="UTF-8" ?>
<class>
#set ($listOfFields = $class.getFields())
#foreach ($field in $listOfFields)
   <field>
      <type>$field.getType()</type>
      <name>$field.getName()</name>
   </field>
#end
</class>
```

Figure 4.13: Velocity template for generating XML

Given the same application model to work on as the JSP file in Figure 4.9, this template would produce the same XML as shown in Figure 4.10. Compared to the JSP template using Java to generate XML, the Velocity template is half as long and more readable. Compared to the JSP template using Tag Libraries to generate XML, the Velocity template is much easier to make since you do not have to worry about implementing tags to extract information from the application model. In general, the Velocity template gives a much better impression of what is generated from the template.

4.7.3 Choosing a code generation technology

In this project Velocity is chosen over JSP for the following reasons:

- It is easy to get started. Open a new file and start to write the template. You do not have to bother setting up a server. Velocity comes in a jar file and only a few class paths needs to be set.
- Once you are started, it is still easy. You do not have to implement your own tags or worry that not doing so you will get messy and less readable code.
- Velocity supports important programming features such as running through a list or using an if-else statement.

As it turned out Velocity also has other features, that makes it an attractive choice, such as 1) You can define local and global macros and 2) Generating nicely formatted source code is not a problem.

4.7.4 More about how Velocity works

When using Velocity in an application the following is generally what is done:

1. Intialize Velocity
2. Create a Context object
3. Add data objects to the Context
4. Choose a template
5. Merge the template and the data to produce an output

It is done like this:

```
...
public class VelocityTest{
  public static void main(String [] args){
    Velocity.init();
    VelocityContext context = new VelocityContext();
    context.put( "name", new String("Velocity") );
    Template template = null;
    try{
      template = Velocity.getTemplate("myTemplate.vm");
      Writer fw = new FileWriter("myFile.txt");
      template.merge( context, fw );
    }
    catch( Exception e ){}
  }
}
```

Figure 4.14: Using Velocity from Java

The concept of the Context is central to Velocity, and is a common technique for moving a container of data around between parts of a system. The idea is that the context is a carrier

of data between the Java layer and the template layer. The programmer will gather objects of various types, whatever the application calls for, and place them in the context. In the template these objects and their methods and properties will become accessible via template elements called references.

As shown in the code snippet in Figure 4.14 a String object is placed in the Context under the name 'name'. By refering to $name in the Velocity template, the value and methods belonging to the String object is accessible.

There is only one drawback in this approach to sharing data. Static methods and static variables are not accessible from the template if there is no instance of the class. Therefore an instance is necessary in order to call static methods or access static variables. As a design strategy one might create a singleton class containing static helper methods and bind this object in the Context.

As depicted in the code snippet in Figure 4.14 the context information and a Writer object are 'merged' and thus produce the output to a file called myFile.txt. The class handling the merge is the Template class. This class is used for controlling all template operations. The class uses a parser created by JavaCC to create an AST from the template information that is subsequently traversed by a Visitor. When calling the merge method, the AST node structure is 'merged' with the context object to produce the final output [25].

4.8 Design of specific features

In this section we will discuss the implementation of features requiring more careful consideration.

4.8.1 Automatic generation of primary keys

One of the requirements to 3E was to provide a service to automatically generate primary keys. There are many strategies to implement such a service. The most obvious one would be to use the database's built-in counter/sequencer to provide primary keys. But in that case one would become dependent on the type of database used, and 3E should be able to produce portable and database-vendor independent EJB components. Another options would be to create a singleton class managing the primary keys. But the problem would be to create an application-wide singleton across a J2EE application. *"A traditional Java singleton (a class which contains a synchronized static instance of itself) only guarantees one instance per classloader, and a typical J2EE server will contain multiple running classloaders per VM"* [13, p. 112]

Various design patterns and implementation strategies for automatic generation of primary keys are suggested in [13]. The book covers three different best practices of generating primary keys, but two of them requires database querying. The third approach is a *universally unique identifier* (UUID) for EJBs, also known as a *globale unique identifier* (GUID). *"A UUID is a primary key encoded as a string that contains an amalgamation of system information that makes the generated UUID completely unique over space and time..."*[13, p.113]

The book provides an algorithm and an implementation strategy for such a service, without the need for database queries or an application-wide singleton. The service can be implemented in two ways: As a Java singleton or as a Session Bean. We have chosen the singleton implementation due to performance since the implementation using a Session Bean will create a lot of overhead due to security checks and Session Bean creation.

The universally unique identifiers generated by the service will be a 128 bit String consisting of 32 digits. The string will be composed as follows:

1. Digits 1–8 are the hex-encoded lower 32 bits of the System.currentTimeMillies() call.
2. Digits 9–16 are the hex-encoded representation of the 32 bit integer of the underlying IP address.
3. Digits 17–24 are the hex-encoded representation of the call to System.identityHashCode(this), which guarantees to return distinct integers for distinct objects within the same Java Virtual Machine.
4. Digits 25–32 represent a random 32 bit integer generated using the java.security.SecureRandom class. Multiple calls to this method within the same millisecond are guaranteed to be unique.

A primary key consisting of these 32 digits is guaranteed to be unique across all machines in a cluster, across all instances of GUID generators within a Java Virtual Machine, down to the millisecond and even down to the individual method call within each millisecond. Yet there are two theoretical holes in this approach where identical primary keys could be generated. One is if the clock on the server is modified. The other is if two instances of the GUID provider are created at the exact same millisecond on the same machine and the SecureRandom returns the same value in which case two identical primary keys will be generated. But this is rather unlikely to happen.

Besides providing 3E with a GUID provider service, 3E will implement two other options for primary key generation. These do not guarantee to generate unique keys, but are implemented to provide the option of generating java.lang.Integers and java.lang.Long as primary keys.

4.8.2 Supporting compound keys

An Entity Bean can use compound keys and does so by declaring more than one field to be primary keys. Usually a primary key field may not be a primtive type, but primary key fields in a compound key are an exception to this rule.

According to the EJB2.0 specification a primary key class must be implemented by the developer when using compound keys. A primary key class has the following characteristics:

Class requirements: The primary key class must be called the name of the Entity Bean followed by PK, e.g. CoursePK. The class must implement the java.io.Serializable interface.

Constructor requirements The primary key class must have a no-arguments constructor, and can have several overloaded constructors.

Field requirements The fields declared in the primary key class must correspond to the fields declared as primary key in the Entity Bean. All fields must be declared public.

Method requirements The primary key class must have get methods, but set methods are not required since all fields are public. The primary key class must overwrite equals() and hashcode().

When using 3E to generate Entity Beans such a primary key class will automatically be generated if the input file specifies more than one field to be a primary key.

4.8.3 Representing relations between Entity Beans

The generator must support all seven relationship types specified in the EJB2.0 specification [18, chap.7]:

- One-to-one, unidirectional
- One-to-one, bidirectional
- One-to-many, unidirectional
- One-to-many, bidirectional
- Many-to-one, unidirectional
- Many-to-many, bidirectional
- Many-to-many, unidirectional

In TEGA's API a Relation is defined to always have exactly two CMR fields, see Figure 4.15.

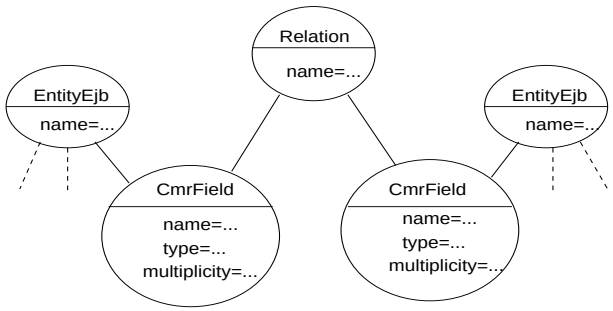

Figure 4.15: Structure of relation in TEGA

Each CmrField object holds information about the multiplicity of the CMR field, the name and type of the CMR field and a reference to the Entity Bean in which the CMR field should appear. This worked well in TEGA, as TEGA at that point did not support unidirectional relations.

In a unidirectional relation, only one of the involved Entity Beans has a CMR field referring to the other Entity Bean. In other words, in a unidirectional relation only one CMR field appears. This made TEGA's way of representing a relation less suitable. A step on the way was to add a boolean variable to the CmrField object defining whether the CMR field was appearing or not. Not appearing meant that the CmrField object was still present in the application model, but the field would not appear, neither in the referred Entity Bean's implementation nor in the deployment descriptor, when the surrounding relation was described. A lot of confusion came from this way of representing a Relation. Instead we were inspired by the way a relation is described in the deployment descriptor. Here a <relation> element always consist of two <role> elements. A <role> element may have a <cmr-field> element, reflecting whether the relation is unidirectional. This description is transferred to our application model: Here a relation is described by a Relation object always referring to exactly two Role objects. A Role object has a reference to a CmrField object, but this may be null reflecting that this role has no CMR field and the relation is unidirectional. Figure 4.16 shows the final representation of a relation in the application model.

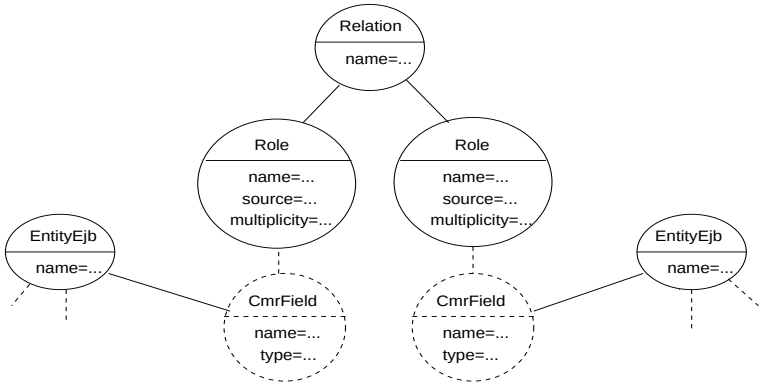

Figure 4.16: Structure of relation in 3E

4.8.4 Generating debug messages

If errors occur when the generated code is compiled, it should be possible to track the error back to the place where the erroneous code is generated. The developer should never correct the error by editing the generated code, but should instead edit the XML input file (so the right code is generated). The optimal solution would be if the compiler gave an error message pointing back to the place in the XML input file, where the code causing the error is defined.

In C and C++ this could be implemented using the #line directive. Unfortunately we have not found a corresponding feature for Java, neither in Javac nor in Jikes.

Instead we will implement a feature making it possible to get what we will call *debug messages* in the generated code. A debug message will be a comment in the code telling were this code is defined in the XML input file. An example can be found in Figure 4.17.

```
// From file apps/app8/app.xml, line 97 to 104
public void setName( String firstname, String lastname){
   // From file apps/app8/app.xml, line 100 to 103
   this.setFirstName(firstname);
   this.setLastName(lastname);
}
```

Figure 4.17: Example showing how debug messages will appear in generated code

If the generated code causes errors during compilation, the compiler's error message will point to a line in the generated code. By using the debug messages, the developer should then be able to find out where this erroneous code came from.

To implement this feature the XML Parser used by Castor XML must give its position in the XML document whenever it passes data from the document. Information about the position should be saved together with the passed data so that whenever this data is used, the information about the position in the XML document can be retrieved.

From the Castor property file (castor.properties) we can learn that the XML Parser used by Castor must support the SAX API. Usually a SAX Parser has a DocumentHandler which receives notification of general document events, such as startElement or endElement. In this way the

DocumentHandler will know whenever the parser starts to parse an XML element and when the same element is finished by the parser. A DocumentHandler usually has a DocumentLocator. The DocumentLocator keeps track of where the last event in the XML document occurred. For instance, it is by means of the DocumentLocator that the parser can tell where it was in the XML document when an exception was thrown. It seems that we can implement the debug messages by using the DocumentHandler to tell, *when* data from an XML element is being passed, and the DocumentLocator to give information about *from where* in the XML document the data is passed.

4.8.5 Read from database

A special feature of 3E is to retrieve metadatainformation from tables and columns in an existing database and generate corresponding Entity Beans.

To do this we need a program that given information about the database can establish a connection through JDBC, access the database and retrieve metadatainformation and map these to our application model. It would not make sense to generate corresponding Entity Bean classes from the database information since no methods are defined. So instead of generating java files, we have chosen to generate an XML input file for 3E. The input file generated can be modified by the user, and methods and other information can be changed or added.

The information necessary to establish a connection to a database is given to the program as an XML document using Castor to parse the XML to a ConnectionSetting object. Using JDBC to establish the connection to the database provides us with a standard interface, so we ought not worry about various database implementations.

Instead of designing the functionality of the program from scratch, we have chosen to use parts of the Middlegen tool described in Chapter 3.2.1. The Middlegen tool provides us with the basic functionality of accessing a database, retrieving the metadatainformation, and mapping these to Schema, Table, Relation and Many2ManyRelation objects. See the class diagram in Figure 4.18.

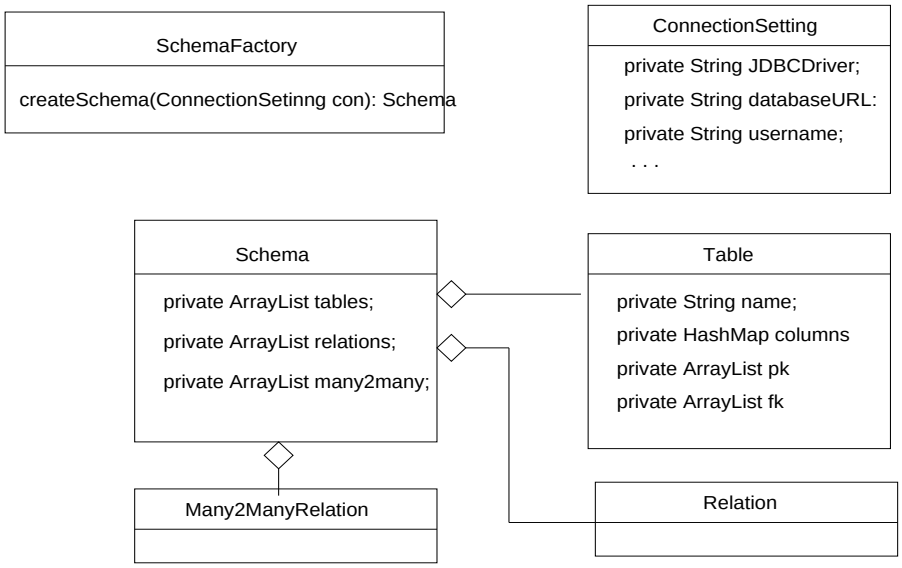

Figure 4.18: Class diagram of MiddleGen's database specific classes

The figure only shows central methods and fields. The class diagram shows the SchemaFactory class which has the method createSchema which takes a ConnectionSetting object as a paramater. The method returns a Schema object. A Schema represents a collection of tables and their relations. A Table object represents database tables and are characterized by a name, primary keys, foreign keys and all columns. Columns are stored in a HashMap to map both the name and datatype of the column.

From the MiddleGen data structure that represents the design of the database it is possible to build an application model representing the Entity Beans, mapping each table to an EntityEjb object, and each column in a table to a Cmpfield and so on. Using Castor the application model is marshalled to XML.

Figure 4.19 shows the complete flow of how the input file for 3E is generated.

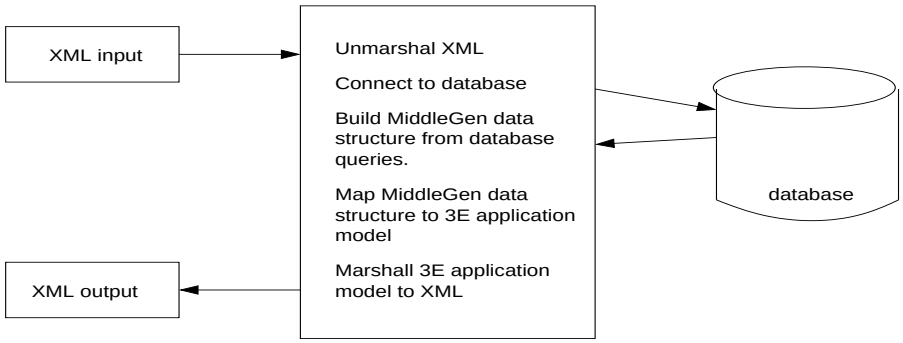

Figure 4.19: How the input file for 3E is generated

Chapter 5

Installation and user manual

5.1 What is 3E?

3E is an open source command line tool for generating CMP Entity Enterprise JavaBeans that conform to the EJB2.0 specification. The 3E tool takes as input an XML file specifying Entity Beans and generates the corresponding bean classes, local interfaces and deployment descriptors.

There are two ways to specify an XML input file. Either do it by hand or provide access to a database, and an XML input file reflecting the tables and relations in the database will be generated. 3E can currently generate EJB applications for JBoss and Weblogic6.1.

With 3E it is possible to costumize Entity Beans by specifying:

- Fields
- Methods
- Relations

Use of compound keys are supported by 3E. If more than one field is specified as a primary key, a primary key class conforming to the EJB2.0 specification will be generated. Automatic generation of primary keys is also supported by 3E. By choosing a primary key to be automatically generated, a global unique identifier class will be generated to provide unique primary keys.

Error messages will be produced if the input file for 3E contains illegal values according to the EJB2.0 specification. If errors occur while compiling the generated code or deploying the EJB application, debug messages can be switched on to help identifying the error in the XML input file.

Generating the input file by accessing a database will produce different outputs depending on the database. 3E has been tested with Oracle8i and MySQL. Oracle8i produces the best result, where tables are represented as Entity Beans and relations between tables are represented as Entity Bean relations. Relations will always be generated as bidirectional relations. The MySQL database does not offer the same possiblities. Only tables will be mapped as Entity EJBs, and no relations will be generated. The generated XML input file can be modified and additional information, such as package name and type of server, must be added before generating the EJB application.

5.2 How to use 3E

This section will present a small example on how to generate an EJB application using 3E. 3E must be installed in order to run this example. How to install 3E is described in Section 5.3.

5.2.1 Specifying an example input file

In Figure 5.1 a simple XML input file for 3E is shown. The input file specifies two Entity Beans, Customer and Address, and creates a one-to-one unidirectional relation between them.

```
<?xml version="1.0"?>
<model xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="../src/schemas/ejbSchema4_0.xsd"
   package-name="dk.itu.bogm">

   <entity-ejb name="Customer">
      <cmp-pk-field type="java.lang.String" name="pk" auto-generate-primary-key="true"/>
      <cmp-field type="java.lang.String" name="firstName"/>
      <cmp-field type="java.lang.String" name="lastName"/>
      <create-method name="createCustomer"/>
   </entity-ejb>

   <entity-ejb name="Address">
      <cmp-pk-field name="pk1" type="java.lang.Long" auto-generate-primary-key="true"/>
      <cmp-pk-field name="pk2" type="java.lang.String" auto-generate-primary-key="true"/>
      <cmp-field type="java.lang.String" name="address"/>
      <create-method name="createAddress"/>
   </entity-ejb>

   <relation name="Customer-Address">
      <role source="Customer" name="Customer-has-address" multiplicity-of-source="One">
         <cmr-field name="homeAddress"/>
      </role>
      <role source="Address" name="Address-belongs-to-customer"
                                 multiplicity-of-source="One"/>
   </relation>

   <deployment>
      <type-of-server>jBoss</type-of-server>
   </deployment>
</model>
```

Figure 5.1: The example input file Simple.xml

All Java files generated will be placed in the package dk.itu.bogm due to the package-name attribute on the root element <model>. The Custormer Entity Bean specifies three fields: Two CMP fields and a CMP field as primary key. The field 'pk' sets the optional attribute auto-generate-primary-key to 'true', which means that in the implementation of the bean class the pk field should be assigned an automatically generated value. Furthermore the Customer Entity Bean specifies a no-arguments create method called createCustomer. The Address Entity Bean is basically identical, except it specifies two fields to be primary keys. This will result in 3E generating a primary key class for the Customer Bean.

The example furthermore shows how a relation is specified. The relation Customer-Address is a one-to-one unidirectional relation since only the Customer-has-address role specifies a CMR field and the multiplicity-of-source attribute in both roles is set to 'One'.

Provided that 3E is correctly installed and that you from a command prompt, while standing in the installation directory for 3E, types

```
ant run-Xml2ejb -Dpath="[relative path to Simple.xml]"
```

the following files will be generated:

- Address.java
- AddressHome.java
- AddressBean.java
- AddressPK.java
- Customer.java
- CustomerBean.java
- CostumerHome.java
- GUIDProvider.java
- ejb-jar.xml

The files can be examined in Appendix B.

5.2.2 Using Ant

3E is integrated with Ant and comes with a build file with predefined command line tasks. The tasks are as follows:

- options — lists all tasks defined.
- javadoc — generates 3E Javadoc into the docs directory
- compile — builds 3E into the classes directory
- run-Xml2ejb — given a XML input file an EJB application is generated.
- run-Db2xml — given a XML input file with connection settings and a path to database drivers, an XML input file for 3E is generated.

How to use the built-in commands?

Ant is a command line tool. To use the built-in commands go to the directory where 3E is installed and the build.xml file is located. Type `ant compile` and all 3E Java files will be compiled. By typing `ant options` all options will be listed. The two commands run-Xml2ejb and run-Db2xml need a XML input file which is given as a parameter to the command, e.g:

```
ant run-Xml2ejb -Dpath="<path to the XML input file specifying Entity Beans>"

ant run-Db2xml -Dpath="<path to the XML input file specifying \
connection settings>" -DDBdriver="<path to JDBC drivers>"
```

If you want debug messages in the generated code that point back to the XML input file specifying the Entity Beans, you should add an additional parameter to the Ant command. Just type `-Ddebug true` right after the path to the input file:

```
ant run-Xml2ejb -Dpath="<path to the XML input file>" -Ddebug="true"
```

How to call Ant from another directory?

If 3E is installed in a directory to which you do not have access, you can use 3E from another directory, by specifying the path to the build.xml file.

```
ant -file <your path>3E/build.xml options
```

Where will the generated files go?

The files generated when calling run-Xml2ejb or run-Db2xml will be placed in the same directory as the XML input file specified. Thus when you run the run-Xml2ejb command several Java files and XML files will be added to that directory and when you run the run-Db2xml command an XML file called application.xml will be generated. The application.xml file is an input file for 3E.

5.3 Installing 3E

3E is distributed as a jar file containing all necessary files and packages. Both java and class files are distributed including documentation (Javadoc). Furthermore all tools used by 3E are distributed along with the jar file. Thus included in the jar file are:

- Castor 0.9.4.1-xml
- Ant 1.5
- Velocity 1.3

5.3.1 System requirements

3E has been tested succesfully on Linux RedHat7.3 and Windows NT platforms.
A version of Sun Microsystems J2SE 1.3 or later must be installed.

5.3.2 Installing 3E

3E comes fully distributed with all necessary files and packages included. In order to install 3E, the environment variable JAVA_HOME must be set to point at the directory where J2SE is located. Furthermore you must make sure Ant is installed. Either download Ant1.3 or later from Apache and follow the installation manual or use the version of Ant distributed with 3E.

If you choose to use the version of Ant distributed with 3E, you must set the following environment variables:

Under Unix(bash):

```
export ANT_HOME=/[your path]/3E/lib/ant
export PATH=${PATH\}:${ANT_HOME\}/bin
```

Under Windows:

```
set ANT_HOME=[your path]/3E/lib/ant
set PATH=%PATH%;%ANT_HOME%/bin
```

5.4 Developer's guide

Before generating an EJB application an XML input file specifying Entity Beans needs to be constructed. In the 3E subdirectory `schemas` one can find the XML Schema ejbSchema4_0.xsd, which describes the structure of an XML input file for 3E. 3E will expect the XML input file *only* to contain XML elements described in ejbSchema4_0.xsd. Any kind of XML element *not* described in ejbSchema4_0.xsd will cause 3E to fail. Unknown or badly-spelled XML attributes will not have this effect, but will simply be skipped when the XML input is read by 3E. The best way to ensure that the XML input file will be accepted by 3E is to validate it against the schema ejbSchema4_0.xsd.

A good XML editor will ease the job of creating the XML input file since it will help remember the names of elements and attributes and can also validate the input file against the schema ejbSchema4_0.xsd.

For examples on how to specify the XML input file look in the 3E subdirectory *examples*. Here you will find a complex example describing an imaginary business called Titan Cruise.

5.4.1 Elements in the XML input file

This section describes elements in the XML input file.

The <model> element

The root element of the input file is the <model> element, the schema definition is shown in Figure 5.2.

```
<xs:complexType name="model_type">
   <xs:sequence>
      <xs:element ref="ejb:entity-ejb" maxOccurs="unbounded"/>
      <xs:element ref="ejb:relation" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="ejb:deployment"/>
   </xs:sequence>
   <xs:attribute name="package-name" type="xs:string" use="required"/>
   <xs:attribute name="data-source-jndi-name" type="xs:string"/>
</xs:complexType>
```

Figure 5.2: Schema definition of the <model> element

A model must be provided with a value for the `package-name` attribute. Other attributes are optional. Note that to validate the input file against the schema ejbSchema4_0.xsd, the <model> element, being the root element, should refer to the schema with the `noNamespaceSchemaLocation` attribute, see how in the example input file in Figure 5.1. The <model> element has three children: <entity-ejb>, <relation> and <type-of-server>. At least one <entity-ejb> must be defined, <relation> is optional and exactly one <deployment> must be specified.

The <entity-ejb> element

The element <entity-ejb> defines an Entity Bean, and the schema definition can be seen in Figure 5.3.

```
<xs:complexType name="entity-ejb_type">
   <xs:sequence>
      <xs:element ref="ejb:all-imports" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="ejb:bean-class-imports" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="ejb:local-component-imports" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="ejb:local-home-imports" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="ejb:cmp-pk-field" maxOccurs="unbounded"/>
      <xs:element ref="ejb:cmp-field" minOccurs="0" maxOccurs="unbounded"/>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element ref="ejb:create-method"/>
         <xs:element ref="ejb:find-method"/>
         <xs:element ref="ejb:select-method"/>
         <xs:element ref="ejb:home-method"/>
         <xs:element ref="ejb:private-method"/>
         <xs:element ref="ejb:business-method"/>
      </xs:choice>
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="required"/>
   <xs:attribute name="default-transaction-attribute" type="xs:string"/>
   <xs:attribute name="find-by-primary-key-transaction-attribute" type="xs:string"/>
</xs:complexType>
```

Figure 5.3: Schema definition of the <entity-ejb> element

As shown in Figure 5.3 the entity-ejb element has three attributes, where only `name` is required. The `default- transaction-attribute` is used if all or a majority of methods in the Entity EJB are to have the same transaction level. By specifying the default-transaction-attribute it is not necessary to specify the transaction level for every method defined in for the Entity EJB. If no default-transaction-attribute is specified, the default transaction level will be 'Required'. The last attribute is the `find-by-primary-key-transaction-attribute`. Since the method findByPrimaryKey according to the EJB2.0 specification always must be defined, 3E always generates such a method, and the method need not be defined in the input file. To provide the findByPrimarykey method with a transaction level, use the attribute on the <entity-ejb> element.

Besides the attributes, the <entity-ejb> element has several children. The first four have to do with importing packages and are optional. To specify a package that both interfaces and bean class must import, use the <all-imports> element. To specify a package that only the bean class needs, use the <bean-class-imports> element and so forth.

The remaining children have to do with fields and methods of the Entity Bean and will be described in the following sections.

The <cmp-pk-field> and the <cmp-field> elements

The <cmp-pk-field> element is used to specify the primary key, therefore at least one <cmp-pk-field> must be specified. If the primary key is a compound key, a <cmp-pk-field> element is used

to specify each of the fields that forms the compound key. The <cmp-field> element is used to define ordinary CMP fields.

The schema definition of the <cmp-pk-field> element is found in Figure 5.4.

```
<xs:complexType name="cmp-pk-field_type">
   <xs:attribute name="type" type="xs:string" use="required"/>
   <xs:attribute name="name" type="xs:string" use="required"/>
   <xs:attribute name="auto-generate-primary-key" type="xs:boolean"/>
   <xs:attribute name="is-get-method-visible" type="xs:boolean"/>
   <xs:attribute name="transaction-attribute-for-get-method" type="xs:string"/>
</xs:complexType>
```

Figure 5.4: Schema definition of the <cmp-pk-field> element

For the <cmp-pk-field> the attributes `type` and `name` are required. The remaining three attributes are optional. The `auto-generate-primary-key` attribute is a boolean signaling that the generated EJB application must provide the primary key. If this attribute is set to 'true', then the `type` attribute must hold either java.lang.Integer, java.lang.Long or java.lang.String, since these are the only primary key types 3E will garentee automatic generation of.

The next attribute, the `is-get-method-visible` has to do with the interfaces. A field in the bean class is declared by abstract get and set methods, but should these methods be reflected in the component interface? The default value is 'true', so use this attribute only if you wish not to display the get method in the component interface. Note that a primary key may never be changed and therefore a set method for the primary key may never be found in the component interface. This is the reason for not having a `is-set-method-visible` attribute on the <cmp-pk-field> element. The last attribute is the `transaction-attribute-for-get-method`. Here you can specify any transaction level wanted for the get method, the default value being 'Required'.

The <cmp-field> element is very similar to the <cmp-pk-field> element. Yet, unlike primary key fields, ordinary fields can have both get and set methods in the interfaces. Therefore, the <cmp-field> element has additional attributes to set the visibility and transaction level for set methods as well.

A few rules to remember when specifying cmp-pk-fields and cmp-fields:

- When specifying the type of a cmp-pk-field, always specify the type with a full class name, for instance java.lang.Integer instead of just Integer. An exception to this rule is if the cmp-pk-field is part of a compound key, thus a full class name is only needed for types not in the java.lang package.
- java.lang.String, java.lang.Long and java.lang.Integer are the only types for cmp-pk-fields that can be automatically generated.
- A full class name is not necessary when specifying the type of a cmp-field. Yet, if the type is not part of the java.lang package, remember to import additional packages for the Entity Bean.

The <...-method> elements

All standard callback methods, all get and set methods and the findByPrimaryKey method are automatically generated and need not be specified in the input file. On the other hand, all other kinds of methods must be defined using the appropiate <...-method> element. Thus, create methods are defined by using the <create-method> element, home methods by using the <home-method> element and so on. Any number of all kinds of methods can be specified for an Entity Bean. Figure 5.5 shows the schema definition of a business method.

```
<xs:complexType name="business-method_type">
   <xs:sequence>
      <xs:element ref="ejb:parameter" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="ejb:application-exception" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="ejb:method-body"/>
   </xs:sequence>
   <xs:attribute name="return-type" type="xs:string" use="required"/>
   <xs:attribute name="name" type="xs:string" use="required"/>
   <xs:attribute name="transaction-attribute" type="xs:string"/>
</xs:complexType>

<xs:complexType name="parameter_type">
   <xs:attribute name="type" type="xs:string" use="required"/>
   <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
```

Figure 5.5: Schema definition of the <business-method> element and the <parameter> element

A <business-method> element has three attributes. The `return-type` attribute is used to specify the return type of the method and the `name` attribute is used to specify the name of the method. Both are required. The optional `transaction-attribute` attribute is used to specify the transaction level for this method.

Along with the attributes the business methods have three child elements. The <parameter> element is optional and is used to specify parameters for this method. As seen from the schema, a business method can have zero or more parameters. The definition of the <parameter> element is shown in Figure 5.5. It has two attributes, `name` and `type`, which both are required.

The second element is the <application-exception> element. This element is optional and specifies which application specific exceptions this method might throw. Either make sure to import the package where the specified exception is defined or write the full class name for the exception.

The last element is the <method-body> element. The element is required and can only appear once. This element is used to specify the implementation of the method and must be written as Java code. Remember to use CDATA sections if the Java code contains characters that needs to be escaped from XML syntax, such as "<" and ">".

The other method elements are basically identical to the <business-method> element. The query methods do not have <method-body> elements but <ejb-ql> elements. Some methods have the <application-exception> element and some do not. The only method element that is different from the others is the <create-method> element, which is why the schema definition of this method is depicted in Figure 5.6.

```
<xs:complexType name="create-method_type">
   <xs:sequence>
      <xs:element ref="ejb:field-to-instantiate" minOccurs="0"
                                                 maxOccurs="unbounded"/>
      <xs:element ref="ejb:ejb-create-method" minOccurs="0"/>
      <xs:element ref="ejb:ejb-post-create-method" minOccurs="0"/>
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="required"/>
   <xs:attribute name="transaction-attribute" type="xs:string"/>
</xs:complexType>

<xs:complexType name="ejb-create-method_type">
   <xs:choice minOccurs="0">
      <xs:element ref="ejb:method-body"/>
   </xs:choice>
</xs:complexType>

<xs:complexType name="ejb-post-create-method_type">
   <xs:choice minOccurs="0">
      <xs:element ref="ejb:method-body"/>
   </xs:choice>
</xs:complexType>
```

Figure 5.6: Schema definition of the <create-method> element

A create method has three child elements. The first is the <field-to-instantiate> which is basically the same as a parameter, but instead of specifying parameters write the name of the field that needs to be instantiated. Just make sure that the name corresponds to a name of a CMP field specified in the Entity Bean. The two remaining children are the <ejb-create-method> element and the <ejb-create-method> element which both have a <method-body> element. The <method-body> element should only contain Java code and need not contain code that instantiates the fields, since this will be automatically generated by 3E.

A few rules to remember when specifying methods:

- When specifying parameters for find and select methods the types must be given with a full class name, e.g. java.lang.String.
- In all other cases the type for parameters and fields does not need a full class name. If the type is not in the java.lang package the package should be imported.
- The values for transactions attributes are: 'Required', 'RequiresNew' and 'Mandatory'. Remember the default value is 'Required'.
- Remember to use CDATA section on Java code and EJB QL.
- If Java code inside a <method-body> element includes use of classes not included in the java.lang package the Entity Bean must import the additional packages.
- In generated code, Java code specified within a <method-body> element will appear with the same formatting as in the input file concerning line breaks etc.

The <relation> element

The second child element of a model is the <relation> element, see the schema definition in Figure 5.7.

```
<xs:complexType name="relation_type">
   <xs:sequence minOccurs="2" maxOccurs="2">
      <xs:element ref="ejb:role"/>
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="role_type">
   <xs:sequence minOccurs="0">
      <xs:element ref="ejb:cmr-field"/>
   </xs:sequence>
   <xs:attribute name="source" type="xs:string" use="required"/>
   <xs:attribute name="name" type="xs:string" use="required"/>
   <xs:attribute name="multiplicity-of-source" use="required">
      <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:enumeration value="One"/>
            <xs:enumeration value="Many"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:attribute>
</xs:complexType>
```

Figure 5.7: Schema definition of the <relation> element and the <role> element

The <relation> element is used to define relations between Entity Bean. It must be provided with a `name` and must specify exactly two <role> elements. The <role> elements have three attributes which all are required. The `source` attribute is used to refer to the Entity Bean involved in this relation, the `name` attribute is the name of this role, and the attribute `multiplicity-of-source`, specifies whether the source Entity Bean is attached with multiplicity 'One' or 'Many'.

The <role> element has one child element, the optional <cmr-field> element. A CMR field is much like ordinary fields, except the type of the field is not required. If the CMR field is to represent a single Entity Bean, the type need not be specified. If instead the CMR field is to represent a collection of Entity Beans, the type must be either java.util.Collection or java.util.Set. Take a look at Figure 5.9 which specifies the one-to-many unidirectional relation depicted in Figure 5.8.

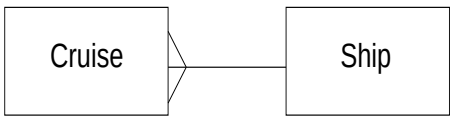

Figure 5.8: A one-to-many relation between ship and cruise

```
<relation name="Cruise_Ship">
   <role source="Cruise" name="Cruise-belongs-to-ship" multiplicity-of-source="Many">
      <cmr-field name="ship"/>
   </role>
   <role source="Ship" name="Ship-has-cruises" multiplicity-of-source="One"/>
</relation>
```

Figure 5.9: A one-to-many unidirectional relation specified in XML

In the example a ship and a cruise form a relation, where a ship can have many cruises and a cruise only belongs to one ship. The name of the relation is Cruise_Ship. The relation defines two roles. The first role is called Cruise-belongs-to-ship, and the source Entity Bean is Cruise. The Cruise Bean is attached to the Ship Bean with multiplicity 'Many': A ship can have many cruises. The other role is called Ship-has-cruises and specifies that this role involves the source Entity Bean Ship. The Ship Bean is attached to the Cruise Bean with multiplicity 'One': A cruise belongs to one ship. Furthermore, the role Cruise-belongs-to-ship specifies a CMR field. Since Ship is attached with a multiplicity of one, the type of the CMR field need not be specified.

Remember when specifying relations:

- When specifying the type of a CMR field used to hold a collection, the type must be java.util.Collection or java.util.Set.

The <deployment> element

The remaining child element of a model is the <deployment> element. The <deployment> element has one child, the <type-of-server> element, where the name of the server to use is specified. At the moment 3E supports only WebLogic6.1 and JBoss, and specifying any other server will result in errors. If WebLogic6.1 is chosen as type of server, the attribute `data-source-jndi-name` on the <model> element must be specified since this information is needed to generate the WebLogic specific deployment descriptor. Also, if a WebLogic6.1 server is used, note that the generated weblogic-cmp-rdbms-jar.xml will specify names on database tables that should match the table names in the database used by the server.

Some rules to remember when specifying deployment:

- If type of server is WebLogic6.1, the attribute `data-source-jndi-name` on the <model> element must be specified

5.4.2 Exceptions when running 3E

Again, note that 3E expects the XML input file to contain only elements described in the schema ejbSchema4_0.xsd. Still, as 3E also checks data read from the input file against the EJB2.0 specification and furthermore has some requirements of its own, exceptions can occur caused by missing or illegal data. The following list shows what is checked by 3E:

* In general:

    - Data required from the developer is present (not null)

- String values required from the developer is not empty Strings

* About Models

  - A model does not contain two Entity Beans or two relations with the same name

* About Entity Beans

  - At least one Entity Bean is specified
  - All Entity Beans have a primary key
  - If the primary key is not a compound key, the primary key type may not be a primitive type
  - If the primary key is auto-generated by 3E, the type must be java.lang.Long, java.lang.Integer or java.lang.String
  - An Entity Bean does not have two or more fields with the same name

* About methods

  - The name of a create method always start with 'create'
  - If the primary key is auto-generated by 3E, the primary key field may not be among the fields initialized by a create method
  - Fields to be initialized by a create method always match a field in the Entity Bean (This means that a field with the exact same name must be found in the Entity Bean)
  - The name of a home method never starts with 'ejbHome', 'create', 'find' or 'remove' (is case-sensitive)
  - The name of a business method does not start with 'ejb' (case sensitive)
  - The name of select method always start with 'ejbSelect' (case sensitive)
  - The name of find method always start with 'find' (case sensitive)
  - A find method named 'findByPrimaryKey' is not specified by the developer (case sensitive)
  - All specified transaction-attributes are valid (Currently this means that a transaction-attribute must be 'Required', 'RequiresNew' or 'Mandatory') (case-sensitive)

* About relations

  - A relation always has exactly two roles with names that are unique within the relation
  - A role always refers to an existing Entity Bean
  - The multiplicity of each role in a relation is always exactly 'One' or 'Many' (case sensitive)
  - The type of a collection based CMR field is always java.util.Collection or java.util.Set

* About the type of server

  - The type of server is always 'JBoss' or 'WebLogic6.1' (not case sensitive)
  - If type of server is 'WebLogic6.1', a dataSourceJndiName must be specified by the developer

If erroneous data are found this will cause an IllegalInputException with a message describing the problem and indicating a solution. For instance, if the server is specified to be WebLogic6.1 but no data-source-jndi-name is specified, the message would be '`dataSourceJndiName in Model must not be null`', indicating that a dataSourceJndiName needs to be specified in the input file.

5.4.3 Specifying an XML input file for database connection

Specifying the input file for running the `run-Db2xml` command is quite simple. An input file could look like figure 5.10:

```
<?xml version="1.0"?>
<connection-settings
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../src/schemas/connection.xsd">

   <JDBC-driver>oracle.jdbc.driver.OracleDriver</JDBC-driver>
   <database-URL>jdbc:oracle:thin:@oracle.it-c.dk:1521:odb
   </database-URL>
   <username>bogm</username>
   <password>*********</password>
   <schema>BOGM</schema>
   <catalog></catalog>
</connection-settings>
```

Figure 5.10: An input file for running run-Db2xml

As Figure 5.10 shows, the name of the JDBC driver must be specified as well as the URL for the database. User name and password must be specified to provide access, but both schema and catalog can be left empty.

For further documentation please consult the connection schema which is found in Appendix F.

The file generated by running the run-Db2xml command can be modified and supplied with aditional information.

Chapter 6

Technical description

If the content of 3E.jar is extracted, the following directories are found:

`classes`	mapping files, a property file needed by Velocity and Java class files
`docs`	documentation of all Java code
`examples`	XML input file examples
`lib`	tools used by 3E, i.e. Ant, Castor and Velocity
`src`	all source code for 3E

This chapter describes the source code, the content of `src`. Details that can easily be learned by reading the source code will be left out. After reading this chapter a developer should be able to change or extend 3E.

6.1 Structure of 3E source code

Figure 6.1 shows the overall structure of 3E.

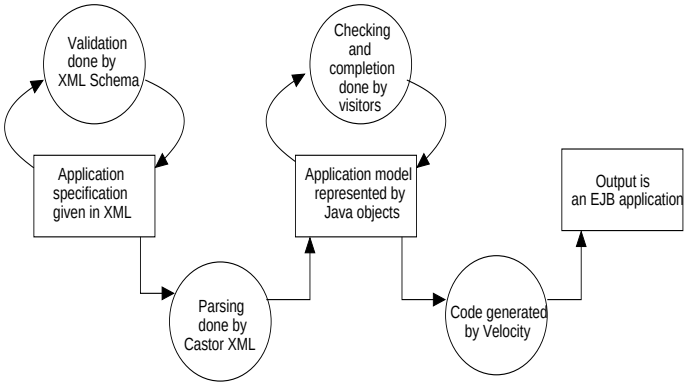
Validation done by XML Schema
Checking and completion done by visitors
Application specification given in XML
Application model represented by Java objects
Output is an EJB application
Parsing done by Castor XML
Code generated by Velocity

Figure 6.1: The overall structure of 3E

The input to 3E is an application specification given in XML. An XML Schema is used to validate the input. Afterwards the open source tool Castor XML is used to pass data from the XML input to an application model implemented in Java. The data in the application model are checked and completed by running through the application model several times; The traversing

is implemented using the visitor pattern. Finally Velocity templates are used to generate the Java and XML output. All XML Schemas, all mapping files needed by Castor XML, all Velocity templates and all Java source code implementing the application model and the visitors are found in the `src` directory, see Figure 6.2.

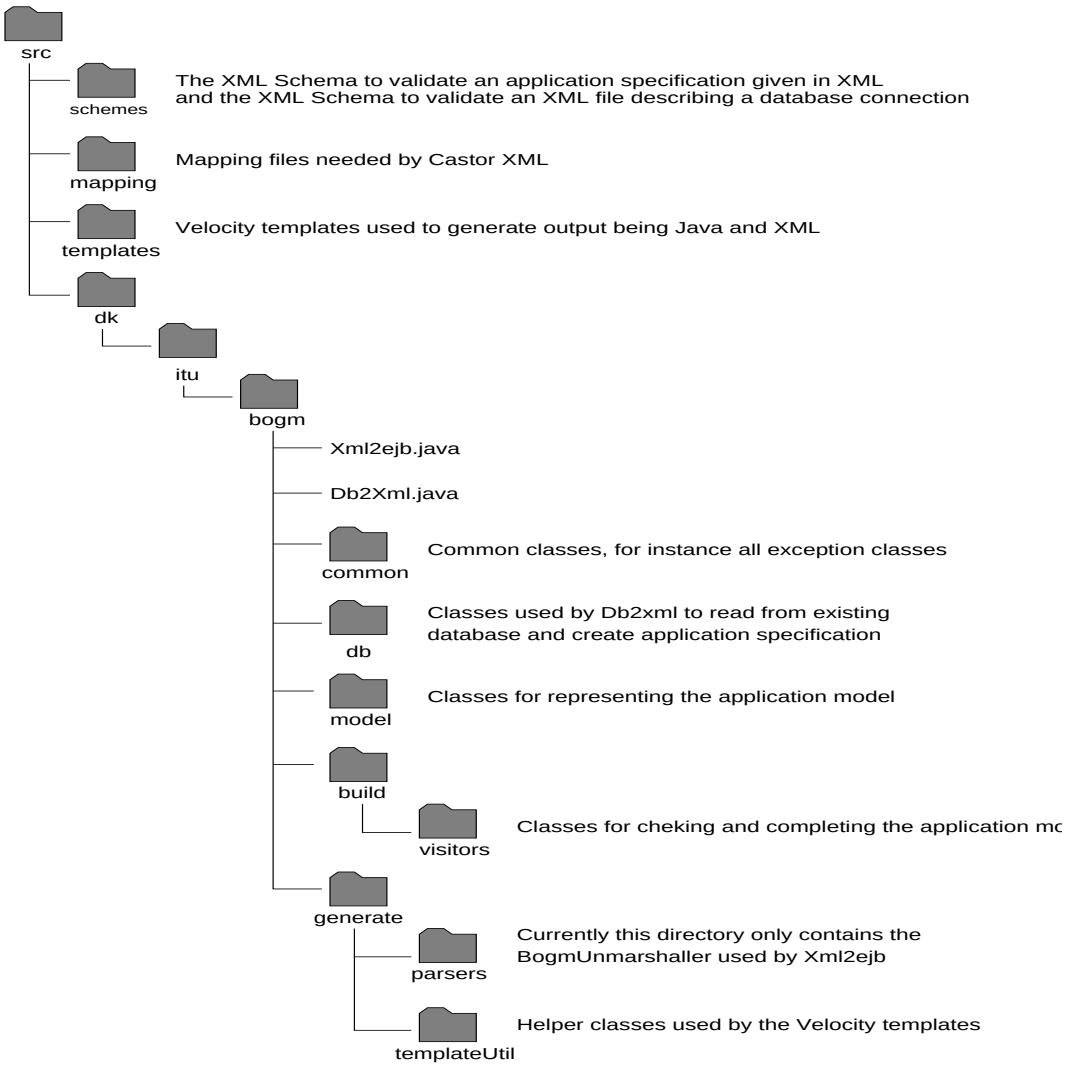

Figure 6.2: Content of the `src` directory

The directory `schemas` holds all XML Schemas, that is, the XML Schema to validate an application specification (ejbSchema4_0.xsd) and the XML Schema to validate an XML file describing the connection settings for a database (connectionSchema1_0.xsd). The `mapping` directory contains all mapping files needed by Castor XML to map data from the XML input file to the application model. The directory `templates` contains all Velocity templates and a property file. The directory `dk.itu.bogm` is the root directory for all Java source code. Located beneath this is:

`Xml2ejb`	Main class for generating an EJB application from an application specification given in XML
`Db2xml`	Main class for an generating application specification in XML based on information from an existing database
`db`	Classes for reading information from an existing database and creating the corresponding input file. Used by Db2Xml
`common`	Contains all common classes, for instance the implementation of 3E exception classes
`model`	Classes for representing an application model
`visitors`	Classes for cheking and completing the application model. As the traversing is implemented using the visitor pattern,we will refer to these classes as *the visitors*
`parsers`	Currently only one class is located in this directory, that is the BogmUnmarshaller used by Xml2Ejb
`templateUtil`	Currently only one class is located in this directory, that is the Utility class, which is a helper class used by the Velocity templates

6.2 XML Schema for an application specification expressed in XML

This section describes ejbSchema4_0.xsd located in the directory `src/schemas` and found in Appendix F. The XML Schema defines the language for expressing an application specification in XML and can also be used to validate the specification.

Every element described in ejbSchema4_0.xsd corresponds to a source class in the package dk.itu.bogm.model and vice versa. E.g. the class Model corresponds to the element <model>, class EntityEjb to the element <entity-ejb> and so on. The schema describes what each element in the schema consists of, e.g. that a <model> consists of a package-name, an optional data-source-jndi-name, at least one <entity-ejb> element, zero or more <relation> elements and exactly one <deployment> element.

Note that the default value for the attributes minOccurs and maxOccurs is 1. Also, note the use of the elements <sequence> and <choice>: A sequence defines a list of sub-elements which must all appear exactly once and in the specified order. A choice defines a list of sub-elements from which exactly one element must be chosen. A <sequence> and a <choice> element can both define a minOccurs and a maxOccurs attribute. The attributes will specify how many times a sequence of elements specified in a <sequence> element can occur, or how many times one can chose an element among the elements specified in a <choice> element.

The schema reflects some decisions about what is a legal specification of an EJB application. E.g. a minimal valid XML input file contains a model with a package name, one named Entity Bean with a primary key (a cmp-pk-field), and information about which server the EJB application is supposed to be deployed on.

Otherwise decisions reflected in the schema are mainly made with the intention to prevent the

developer from defining faulty or meaningless input, such as defining a business method without defining a method body, or defining a relation with only one role. Still elements and attributes can be left empty, but this will be caught later by the checks made by the visitors. The schema could be more strict by using the restriction tag to define legal values for attributes such as the transaction attribute on methods. Mostly such restrictions are avoided in the schema and instead it is left to the visitors to implement a check of whether given values are legal. Decisions about what is legal could change according to what the generator should support, and by letting all those decisions be reflected in one place only, it will be easier to change the decisions later. Since the developer can create an application specification using the API for the application model instead of an XML input file, all decisions are placed in the visitors instead of in the schema.

The XML Schema should always reflect what data 3E expects to find in an application specification. Thus the schema should be extended to define additional elements, if this is required by an extended version of 3E.

6.3 Mapping files needed by Castor XML

Mapping files are XML documents used by Castor XML [3] to parse XML to Java objects. Mapping files are located in the `src/mapping` directory and can be found in Appendix F. An example of a mapping file was shown in Figure 4.5 on page 31 and will not be repeated here.

The mapping directory contains five mapping files. The file ConnectionMapping.xml contains information on how to map an XML document that has been validated against the connectionSchema1_0.xsd, to a dk.itu.bogm.db.util.ConnectionSetting (Appendix F) object.

The four remaining files are used when mapping from an application specification given in XML to an application model. Instead of producing one large mapping file, the XML mapping files have been divided into smaller files referring to each other. The main file is the ModelMapping.xml. From this file there are references to the other mapping files used by Castor XML to perform the XML parsing. See figure 6.3.

```
.....
<mapping>
   <include href="mapping/EntityEjbMapping.xml"/>
   <include href="mapping/RelationMapping.xml"/>
   <class name="dk.itu.bogm.model.Model">
      <map-to xml="model"/>
      <field name="packageName" type="java.lang.String">
         <bind-xml name="package-name" node="attribute"/>
      </field>
   ...
</mapping>
```

Figure 6.3: Code snippet from ModelMapping.xml

The mapping files describe how each element and attribute defined in the ejbSchema4_0.xsd should be mapped to the application model. If changes or modifications are applied to the ejbSchema4_0.xsd or in the application model, these should be reflected in the mapping file.

6.4 Implementation of the application model

The implementation of the application model is located in the package dk.itu.bogm.model which can be found in Appendix F. The package contains classes corresponding to every element described in the XML schema. E.g. the <entity-ejb> element has a corresponding class EntityEjb and the <cmp-field> element has a corresponding class CmpField. Super classes are provided for classes with similarities. E.g. the classes CmpField, CmpPkField and CmrField all inherit from the abstract class Field and the classes representing the various types of methods inherit from the abstract class Method.

All classes used to implement the application model have fields that correspond the properties of the elements in the XML schema ejbSchema4_0.xsd. Furthermore the classes have additional fields to make traversing in the application model easier. Besides the fields, the classes are fairly empty. The basic content of each class is:

- All classes implement the interface Named and extend the abstract class Origin.
- A no-arguments constructor which is required by Castor XML.
- Public get and set methods, which are required by Castor XML. The set methods do not perform any validation of the input as it would normally be done to provide solid encapsulation. All validation is performed by the Visitors.
- An accept method as part of the visitor pattern.
- Some classes have additional helper methods which are used from the Velocity templates.

The interface Named and the abstract class Origin are both located in package dk.itu.bogm.model. The interface Named is described in Section 6.5.2 and the abstract class Origin in Section 6.7

If new classes are added to the application model structure they must all implement the basic content as described above.

Besides the classes representing the XML elements, the package dk.itu.bogm.model also contains certain helper classes to simplify traversing the application model from the Velocity templates. They provide a single entry-point to access data stored in the application model. Thus, there is a helper class for each Java file to be generated: class BeanClass, class LocalHomeInterface and class LocalComponentInterface. The classes contain helper methods called from the Velocity templates. An example of such a method is shown in Figure 6.4. The method

```
public ArrayList getAllAutoGeneratedPrimaryKeys(){
   ArrayList genPk = new ArrayList();
   Iterator i = this.getEntityEjb().getCmpPkFields().iterator();
   while(i.hasNext()){
      CmpPkField current = (CmpPkField)i.next();
      if(current.getAutoGeneratePrimaryKey())
         genPk.add(current);
   }
   return genPk;
}
```

Figure 6.4: Code snippet from the helper class BeanClass

returns a list containing only CmpPkFields that are specified to be automatically generated.

If an extension is added to 3E and new files are to be generated it is a good design strategy to provide helper classes. The single entry-point to access of data in the application model makes the Velocity templates well-arranged.

6.5 Visitors for checking and completing the application model

The purpose of the visitors are to run through the application model five times. Each time some data are checked, new data are added or cross references are set and checked. After these five passages the application model should be consistent and complete — ready to be used for the final code generation. Each passage is implemented by a visitor. The five visitors have a common super class as they all traverse the application model in the same order. All visitor classes are shown in Figure 6.5 and implementations are found in Appendix F.

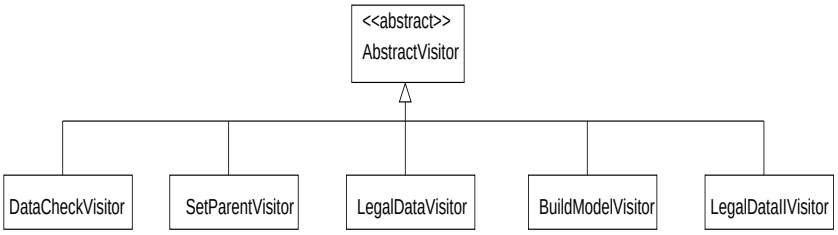

Figure 6.5: The visitors

The abstract super class AbstractVisitor implements the traversal of the application model by calling the accept method on all children of each element from the visit method of the element. Figure 6.6 shows how this is done for an Entity Bean.

In addition to a visit method for each kind of element in the application model, AbstractVisitor also implements three static helper-methods discussed later. Each of the five concrete visitors implement only visit methods for elements that the visitor needs to do something to. For instance, the LegalDataVisitor implements visit methods for Model, Deployment, CmpPkField, Relation, Role and CmrField, since these are the elements checked or modified by the LegalDataVisitor. The last statement in all visit methods in all concrete visitors is a callback to the corresponding visit method in AbstractVisitor. This callback is what ensures that all children of the currently visited element will also be visited.

6.5.1 The five concrete visitor classes

Each of the five concrete visitors perform a separate operation reflected in the name of the visitor. Thus the DataCheckVisitor checks that the expected data are present and valid, meaning that, e.g. the String variable holding the name of an Entity Bean should not be an empty String. The documentation for each visitor gives a description of the operation performed by this visitor. As a starting point there were three operations that should be taken care of by the visitors:

- Data checking: String variables should hold legal values and certain references must be present (not null)

```
...
   public void visitEntityEjb(EntityEjb e)
      throws IllegalInputException, VisitorErrorException {

      // Call accept-method on all children
      Iterator i = e.getAllMethods().iterator();
      while (i.hasNext()){
         Method method = (Method)i.next();
         method.accept(this);
      }

      i = e.getAllFields().iterator();
      while (i.hasNext()){
         Field field = (Field)i.next();
         field.accept(this);
      }

      e.getLocalHomeInterface().accept(this);
      e.getLocalComponentInterface().accept(this);
      e.getBeanClass().accept(this);
   }
...
```

Figure 6.6: Code snippet from AbstracVisitor implementing traversal of the application model

- Set parent references: The XML input file reflects some parent-child-relations that are reflected by parent-to-child references in the application model after parsing. These relations should also be represented the other way by setting the corresponding child-to-parent references.
- Legal data: Data must be checked with regard to the EJB2.0 specification

The data-checking and the set-parent operations are straightforward. The legal-data operation is complex as it requires additional cross references to be set, and these cross references should not be set until it has been checked that they would reflect a legal state. For instance, after the set-parent operation, a relation between two Entity Beans is reflected in the application model by a Relation object with two Role objects, each with an optional CmrField object, cf. Figure 4.16 on page 43. A CmrField object should also be attached to the EntityEjb object that it belongs to, but this should not be done until it has been checked wether the relation is legal. After the CmrField has been attached to the EntityEjb, it should be checked that the EntityEjb does not have two fields with the same name since that would be illegal. To keep things as simple as possible, the legal-data operation is split up into the LegalDataVisitor and the LegalDataIIVisitor, while the BuildModelVisitor is made to perform those operations needed in between. Thus, the BuildModelVisitor sets references that depend upon checks done by the LegalDataVisitor, and afterwards the application model is ready for the remaining checks to be performed by the LegalDataIIVisitor. Applied to the sitation outlined above with the Relation and the corresponding CmrFields and EntityEjbs this means that 1) the LegalDataVisitor checks whether the Relation is legal and if it is 2) the BuildModelVisitor sets the corresponding references between CmrFields and EntityEjbs while afterwards 3) the LegalDataIIVisitor checks that the EntityEjbs does not have two fields with the same name.

The visitors should be executed in the order reflected by the list below, as each visitor depends on work done by the previous visitor:

- DataCheckVisitor
- SetParentVisitor
- LegalDataVisitor
- BuildModelVisitor
- LegalDataIIVisitor

Imagine the application model right after the XML input file has been parsed. It contains only parent-to-child references, while other references are still null and some variables are left uninitialized. After the DataCheckVisitor has been executed, all data are checked. After the SetParentVisitor has been executed, child-parent references are set and so forth. Thus after all visitors have been executed, the application model contains all values and references necessary for the code generating process.

Xml2ejb is the main class for generating code from an XML input file. After the XML input has been parsed to the application model, Xml2ejb calls all five visitors to traverse the application model and execute the operations pointed out above. Thus the application model is made ready for the next step: the code generation.

6.5.2 The static helper methods

The AbstractVisitor implements three static helper methods. Two of them, the checkNotNull method and the checkString method, are mainly used for data checking. The third helper method, checkDuplicates, is shown in Figure 6.7.

```
public static void checkDuplicates(ArrayList list, String parent, String children)
   throws IllegalInputException {

   HashSet set = new HashSet();
   Iterator i = list.iterator();
   while (i.hasNext()) {
      String name = ((Named)i.next()).getName();
      if (!set.add(name)) {
         String msg = parent + " has two " + children + " with name " + name;
         throw new IllegalInputException(msg);
      }
   }
}
```

Figure 6.7: Implementation of method checkDuplicates in AbstractVisitor

The method is used to check if e.g. an EntityEjb object contains two fields with the same name, or if a Model object contains two relations with the same name. As this check is needed by many different objects in the application model, we want to implement the method so that it can be used by all of them regardless of the type of objects to be checked. This is reflected by the parameter `list` of type ArrayList, which can contain any kind of objects, e.g. Field objects

or Relation objects. Instead the method expects that objects in the list always implements the interface Named, which defines a getName-method. Using the getName method to get the name of all objects in the list, the check is done by adding the names to a HashSet. The HashSet method `add(Object obj)` returns false if the name is already in the HashSet.

6.5.3 Possible improvements and extensions

Regarding the checkDuplicates method, it could be improved by implementing a check of whether all objects in the list do implement the interface Named. If not, the method should throw an exception letting the developer know that the check could not be performed.

If a developer makes extensions to 3E that requires new checks, these checks are easily implemented by simply implementing an additional visitor. Actually the generator could also be customized by only choosing a sub-set of visitors to be executed on the application model. In the current version of 3E each of the five visitors depend on work done by the previous visitor, thus choosing a sub-set of visitors to be executed might not make much sence. In an extended version of 3E one could imagine new visitors to, e.g. prepare the application model for at specific type of server or to support additional transaction attributes on methods of an Entity Bean and so forth. Thus, choosing a sub-set of visitors to be executed on the application model could customize the generated code.

6.6 Velocity Templates used for code generation

The Velocity templates are located in `src/templates` and can be found in Appendix F. The directory contains Velocity templates to generate code for each kind of file that can be generated by 3E: BeanClassTemplate.vm, LocalHomeInterfaceTemplate etc. The templates directory also contains the file VM_global_library.vm that defines global macros used by various templates. Furthermore the directory contains the template generate.vm which controls the generation of files. For each EntityEjb specified in the application model a component interface, a home interface and a bean class will be generated, and depending on the type of server specified the corresponding deployment descriptors are generated. The property file configure.properties contains information on where the Velocity templates can be found by 3E while running. The property file is also located in the classes directory since this is the default directory where 3E will look for a property file. Each time the Ant task *compile* has run, the classes directory is deleted and reconstructed thus the property file in the templates directory is copied to the classes directory. Changes added to the property file should thus be added to the property file in the `src/templates` directory.

6.6.1 The Velocity language

This section will present an introduction to the basics of the Velocity language.

Variables in Velocity are either specified in the Velocity Contex (see Section 4.7.4) or they are specified within the Velocity template such as illustrated in Figure 6.8.

```
#set ($foo = "Hello world")
```

Figure 6.8: How variables are defined in Velocity.

Velocity variables are always referenced using the $ sign.

Conditional statements are constructed by the use of *if / else / end* as shown in Figure 6.9. The figure also shows that if the value of the variable $foo is null the *else* statement is executed. The #end statement must be included to end the conditional statement.

```
#if ($foo)
     This line is generated if the value of $foo is true,
     or if the variable $foo is assigned a value other than false.
#else
     This line is generated if the value of $foo is false,
     or if the variable $foo is null.
#end
```

Figure 6.9: How conditional statements used in Velocity.

Loops are used as shown in Figure 6.10. The loop will run through every element in the variable $productList and bind each element to the variable $product. The value of $productlist can be any type of collection and it is not necessary to perform down casts to call methods which are not defined in the Object class.The #foreach statement must be ended with #end.

```
#foreach ($product in $productList)
     $product.getName()
#end
```

Figure 6.10: How conditional loops used in Velocity.

6.6.2 Template layout and formatting of generated output

The layout of the templates determines the formatting of the generated code. By left justifying all Velocity functions, the formatting of the remaining template is reproduced in the generated output. Figure 6.11 shows how the generation of abstract get methods will be nicely formatted

```
public abstract class $beanClass.getName() implements EntityBean{
#set( $ListOfFields = $beanClass.getFields() )

// GET'ers
#foreach ( $field in $ListOfFields )
       abstract public $field.getType() \
                   get$util.firstCharAsCapital($field.getName())();
#end
```

Figure 6.11: Formatting of abstract get methods. A line break is added in the template due to layout of this report. This line break would also be reflected in the generated code.

by providing space characters in the code to be generated and making no formatting of the Velocity functions themselves. To view the generated output see for instance CustomerBean.java in Appendix B.

6.6.3 Velocity macros

In Velocity it is possible to define macros. A macro can be defined within a template, and must be specified before it is used. A macro defined in a template is not accesible from other templates. The file VM_global_library.vm defines global Velocity macros used in various templates.

An example of such a macro is the separateParameters macro Figure 6.12 which is used to make commaseparated lists. Commaseparated lists are necessary when generating parameter lists in method declarations.

```
#macro( separateParameters $list $sep )
#foreach( $element in $list )#if( $velocityCount > 1 )$sep#end \
                      $element.getType() $element.getName()#end
#end
```

Figure 6.12: The separateParameters macro. A line break is added in the macro due to layout of this report.

The code snippet shows the definition of the separateParameters macro which given a collection and a some separator (e.g. ','), will take every element in the collection and separate them by the specified separator. The separator sign will *not* be used before the first element, and *not* behind the last, since the value of $velocityCount is increased by one for each element in the list.

6.6.4 The Utility class

The Utility class found in Appendix F contains helper methods needed by the Velocity templates. E.g. the Utility class specifies a `firstCharAsCapital(String s)` method which return a String with the first letter capitalized. This method is used when generating the names of get and set methods to capitalize the name of the fields, e.g. setName. Another method specified in the Utility class is the `runTemplate(String templateName, String fileName, String directory)` which is used whenever it is necessary to run a template from within another template. This method is thus used in the template generate.vm that controls which templates to run.

The Utility class is implemented as a singleton since only one instance is needed as Velocity templates cannot call static methods, as explained in Section 4.7.4 on page 39. During instantiation the static Utility instance will bind itself in the VelocityContext under the key 'util' to make itself reachable from the Velocity templates. Thus all methods defined in the Utility class are available when using the $util variable in the Velocity templates.

6.6.5 How automatic generation of primary keys is implemented

The Velocity template GUIDProviderTemplate.vm will generate a class called GUIDProvider implementing the algorithm described in Section 4.8.1 to automatically generate primary keys for EJB applications. An implementation of the GUIDProvider class can be found in Appendix B.

```
#if($cmpPkField.getType().equals("java.lang.String"))
     $util.runTemplate("GUIDProviderTemplate.vm", "GUIDProvider.java")
     this.set$util.firstCharAsCapital($cmpPkField.getName())( \
                          GUIDProvider.createGUIDProvider().getGUID());
#end
```

Figure 6.13: How the GUIDProvider class is generated. A line break is added in the template due to layout of the report.

As the code snippet in Figure 6.13 shows, a GUIDProvider class is generated by calling the GUIDProviderTemplate.vm each time a CmpPkField has type String (and is set to be automatically generated though this can not be seen from the figure). The next line of code shows how the code for calling the set method of the CmpPkField is generated including how the GUIDProvider class is called to provide a unique String object as parameter to the set method. View the ejbCreateMethod in customerBean.java in Appendix B to see what is generated by the code snippet in Figure 6.13.

Remember that the GUIDProvider class is implemented as a singleton and only one static instance is created, so the createGUIDProvider method call will result in instantiation of a GUIDProvider object only the first time it is called.

There is one drawback in this implementation. If more than one CmpPkField with type String needs to be automatically generated, the GUIDProviderTemplate.vm will be run several times, overwriting the GUIDProvider.java file each time producing the exact same output.

There are two other implementations of automatic generation of primary keys. The implementation of generating primary keys of type java.lang.Integer and java.lang.Long do not guarantee to be unique, but will provide different values for primary keys if the method is called twice within the same millisecond. The implementation uses a java.secure.SecureRandom object to produce random numbers. Figure 6.14 shows the template implementation of the automatic generation of java.lang.Long primary keys. The implementation of automatically generated java.lang.Integer primary keys is basically identical.

```
#if($cmpPkField.getType().equals("java.lang.Long"))
       this.set$util.firstCharAsCapital($cmpPkField.getName())( \
           new Long(new java.security.SecureRandom().nextLong()));
#end
```

Figure 6.14: How java.lang.Long primary keys are generated in the BeanClassTemplate.vm. A line break is added in the template due to layout of the report.

6.6.6 How compound keys are generated

A compound key class is generated by running the PrimaryKeyClassTemplate.vm (The template can be found in Appendix F). This happens when there is more than one CmpPkField specified in the XML input file. The class generated is implemented as described in Section 4.8.2. An overloaded constructor is added to the class as a convenience method to decrease the number of

steps to create a primary key. The hashCode method is implemented by using a bitwise exclusive-or on the hashcode of the properties. When a class property is a primitive, the primitive is wrapped as an object so hashcode() can be called. See Appendix B to view the generated code.

```
   public int hashCode(){
      int hashCode = 0;
#set( $hash = ".hashCode()" )
#foreach ($field in $listOfCmpPkFields)
#if( !$util.isPrimitiveType( $field.getType() ) )
      hashCode = hashCode^$field.getName()$hash;
#else
      hashCode = hashCode^new $util.wrapPrimitive($field.getType())( \
                                              $field.getName())$hash;
#end
#end
      return hashCode;
   }
```

Figure 6.15: How hashCode is implemented in the primary key class. A line break is added in the template due to layout of the report.

The implementation of the hashCode method in the PrimaryKeyClassTemplate in Figure 6.15 also shows a Velocity subtlety. In the third line a Velocity variable $hash is set to '.hashCode()'. This variable is used in line 6 and 9. It is necessary to use $hash instead of writing .hashcode() to prevent Velocity from actually calling the hashcode method at generation-time.

6.6.7 How to create a new template

Creating a new template to extend the 3E generator can be done by creating a .vm file and adding it to the templates directory. For instance if one wanted to extend 3E to generate deployment descriptors for the Oracle application server, all that is needed is a OracleDeployment.vm which then must be called from the file generate.vm when the type of server is set to Oracle.

6.7 Generating debug messages

This section describes the implementation of generating debug messages. A debug message contains information about the origin of data reflected in generated Java code. Thus, to generate debug messages all data must know where they are specified in the XML input file. Section 4.8.4 discussed how this should be obtained during parsing the XML input file to the application model. Thus, the parser must supply information about its location in the XML file as it passes data to the application model. Usually a parser invokes methods on a DocumentHandler each time a new XML element is found or the current XML element is finished. As a DocumentHandler has a DocumentLocator, it is able to tell the parser's current location in the XML file.

6.7.1 The Unmarshaller object

In Castor XML the parsing (or unmarshalling) is taken care of by an Unmarshaller. When the method `unmarshal(InputSource source)` is called on an Unmarshaller object an instance of a concrete XML parser is initialized and this XML parser is used to parse the XML

document provided by the argument `source`. The concrete XML parser is specified in Castor's property file, in 3E Castor uses Apache's Xerces.

In 3E the Unmarshaller object is created in the main class Xml2ejb. The Unmarshaller calls the unmarshal method on the XML input file provided as an argument when running Xml2ejb. The Unmarshaller uses a DocumentHandler, in Castor XML called an UnmarshalHandler. The UnmarshalHandler is attached to the XML parser after this has been initialized in the unmarshal method. Figure 6.16 shows the relation between the different parts involved in the parsing process.

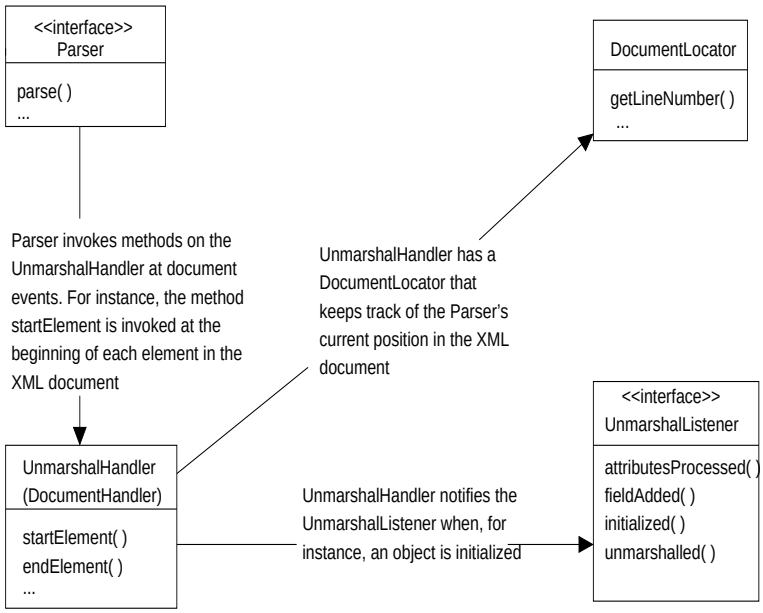

Figure 6.16: Actors in the parsing process

As seen from Figure 6.16 an UnmarshalHandler can have an UnmarshalListener, which will be notified by the UnmarshalHandler when, for instance, an object is initialized or unmarshalled. The UnmarshalListener is an interface defined by Castor. By making an implementation of this interface and attaching this implementation as an UnmarshalListener to the UnmarshalHandler, information can be added to an object when the object is initialized and unmarshalled. For instance, the UnmarshalListener has a method `initialized(Object object)` which is called by the UnmarshalHandler when an object is initialized. The implementation of this method could add information about the position in the XML document to the object.

As we do not want to change in Castor XML directly, we have made our own Unmarshaller, the BogmUnmarshaller, sub-classing the Castor XML Unmarshaller.

6.7.2 The BogmUnmarshaller

In the method `unmarshal(InputSource source)` in Castor's Unmarshaller the XML parser is declared as type Parser. Parser is an interface in the SAX1 API which is deprecated and replaced by the SAX2 API. In SAX1 the XML parser implements the Parser interface and has a DocumentHandler, while in SAX2 the XML parser implements the XMLReader interface and has a ContentHandler. In the unmarshal method of the BogmUnmarshaller the XML parser should not be referenced as type Parser since this would generate a warning telling that the

BogmUnmarshaller is using a deprecated API. To avoid this warning the BogmUnmarshaller uses adapter classes from the SAX API to wrap the XML parser.

In the unmarshal method of the BogmUnmarshaller an UnmarshalListener is attached to the UnmarshalHandler. The UnmarshalListener is implemented as an anonymous inner class. In our implementation of the UnmarshalListener's methods, we use the UnmarshalHandler's DocumentLocator to give the current position in the XML document and add the information about the position to the object. Figure 6.17 shows the implementation of the UnmarshalListener.

```
...
handler.setUnmarshalListener(new UnmarshalListener(){

   public void attributesProcessed(java.lang.Object object) {}

   public void fieldAdded(java.lang.String fieldName,
      java.lang.Object parent, java.lang.Object child) {}

   public void initialized(java.lang.Object object) {
       if ( handler.getDocumentLocator() != null ){
   if (object instanceof Origin) {
       Origin obj = (Origin)object;
       obj.setOriginName( inputFileName );
       obj.setStartLine( handler.getDocumentLocator().getLineNumber() );
   }
       }
   }

   public void unmarshalled(java.lang.Object object) {
       if ( handler.getDocumentLocator() != null ){
   if (object instanceof Origin) {
       Origin obj = (Origin)object;
       obj.setEndLine( handler.getDocumentLocator().getLineNumber() );
   }
       }
   }
});
...
```

Figure 6.17: Implementation of the UnmarshalListener inside the unmarshal-method of the BogmUnmarshaller

As the code snippet in Figure 6.17 implies, the parsed object is cast to an *Origin* object, and on this Origin object the methods setOriginName, setStartLine and setEndLine are called. Origin is an abstract class in package `dk.itu.bogm.model` declaring String originName, int startLine and int endLine, all with corresponding accessor methods. All classes in the application model extend the Origin class, allowing the XML parser to add information about the position in the XML document when an object is parsed. Note that the body of a <method-body> element or an <ejb-ql> element will be parsed to a String object, which will not extend the Origin class. This is why the methods in the UnmarshalListener determines whether the parsed object is an instance of Origin before any information is added to the object. Also, note that if an object in the application model is created by another instance and therefore not parsed from the XML input file, the default value of originName is `[Filename not available]` and for startLine and

endline it is -1.

Both Origin and the BogmUnmarshaller can be found in Appendix F on page 120.

6.7.3 Writing debug messages

Finally the debug messages are generated by letting the main method in Xml2ejb bind a Boolean object called debugValue in the Velocity context. The default value of debugValue is 'false'.

In the Velocity template generate.vm a variable called debug is declared, reflecting the value of the Boolean object debugValue. All remaining Velocity templates generating Java code will examine the debug variable, and when true, debug messages will be included in the generated code. Figure 6.18 shows how this is done.

```
...
// GET'ers
#foreach ( $field in $ListOfFields )
#if( $debug )#generateComment( $field )
#end
abstract public $field.getType() get$util.firstCharAsCapital($field.getName())();
#end
...
```

Figure 6.18: Example showing how a debug message is generated from one of the Velocity templates

Note the `#generateComment` in the code snippet in Figure 6.18: It is a Velocity macro implemented for generating debug messages and is shown in Figure 6.19.

```
#*-----------------------------------------------------------------------------
generateComment macro - given an element this macro generates a comment about
where this element is specified in the input-XML-document.
Usage: To generate comments when debug = true.
-----------------------------------------------------------------------------*#
#macro( generateComment $element )
    // From file $element.getOriginName(), line $element.getStartLine() to \
                                                        $element.getEndLine()
#end
```

Figure 6.19: Implementation of the macro for generating debug messages. A line break is added in the template due to the layout of the report

6.7.4 Improvements

Currently debug messages are generated for all objects in the application model if debug is set to 'true'. The macro `#generateComment` does not differentiate if an object is actually parsed from XML or not. Objects created later (e.g. an EjbCreateMethod object will be created in the DataCheckVisitor if none is specified in the XML input file) will cause debug messages using

the default values from the Origin class, like: `// From file [Filename not available], line -1 to -1`. A check could be made by the macro, so that such debug messages would simply not be generated.

6.8 Exception handling

When exceptions occur using 3E, the error message should guide the user in how to proceed. To ensure meaningful error messages, 3E implements its own exception hierarchy. All exception classes can be found in package dk.itu.bogm.common.exceptions and can be examined in Appendix F. Figure 6.20 reflects the exception hierarchy in 3E. The ExternalErrorException and

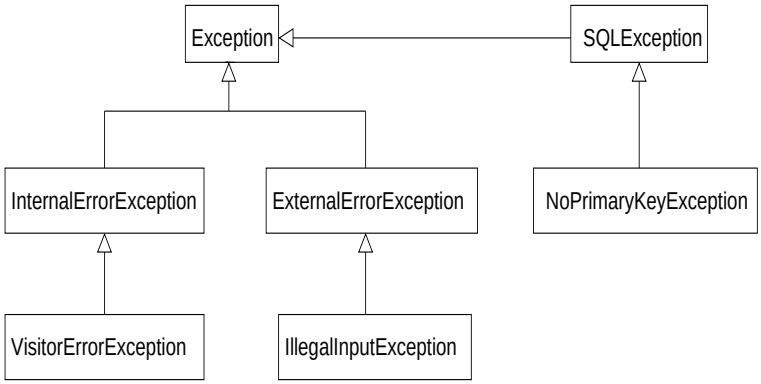

Figure 6.20: The exception hierarchy in 3E

the InternalErrorException are super classes of the remaining exception classes. The ExternalErrorException is the super class for all exceptions caused by something outside 3E, for instance caused by the developer giving erroneous input to the generator. Similarly, the InternalErrorException is the super class for exceptions caused by something inside 3E. A developer using 3E should never get exceptions of type InternalErrorException, as this kind of exception reflects an internal inconsistency in the implementation of 3E and would leave the developer with no options to proceed.

The NoPrimaryKeyException does not conform to the rules pointed out above. It belongs to the part of the generator that looks up information in an existing database; see Section 6.9.

Efforts have been made to generate informative error messages, especially if the exception is caused by input to the generator. E.g. an attempt to describe an Entity Bean `Student` with two fields called `address` would cause an IllegalInputException with the message `Student has two fields with name address`.

6.8.1 Possible improvements

As described in Section 6.7 about implementing debug messages, all objects in the application model contain information about where the object itself is defined in the XML input file. This information could be used to improve error messages, so that for instance the error message written out above would be: `Student defined in line 6 in file input.xml has two fields with name address, one defined in line 9 and the other in line 14`.

6.9 Generating an input file for 3E from database

This section will cover the implementation of generating the XML input file by accessing a database. The structure and flow of the program was presented in Section 4.8.5 on page 4.8.5. This section will present the DatabaseMetadata object from the java.sql library and present the SchemaFactory class that performs all database querying. Finally the conversion of SQL types to Java types is presented.

6.9.1 DatabaseMetadata objects

The generation of the XML input file for 3E is implemented by use of the java.sql.DatabaseMetadata interface. A DatabaseMetadata object is retrieved from a Connection object and provides comprehensive information on the design of the database.

Tables are located be calling the method getTables on the DatabaseMetadata object. This will return a ResultSet containing information on each table in the database or a schema within the database. The ResultSet will also contain information on views and system tables. When table names are located it is possible to query the DatabaseMetadata object for information on each table, retriving information such as column names and datatypes, primary and foreign keys. If no primary key is identified 3E will throw a NoPrimaryKeyException. Relations between tables are located by calling the getCrossReferences method on the DatabaseMetadata object.

In the Java1.3 API on the DatabaseMetadata interface it is stated that: *"If a given form of metadata is not available, the methods of the DatabaseMetadata object should throw an SQLException."* [16] Since Sun Microsystems only provides the interface, not the implementation, they can not guarantee a SQLException to be thrown. In the MySQL implementation of the DatabaseMetadata interface, no metadata on crossreferences is available. Yet the methods do not throw SQL Exceptions, just returns an empty resultSet. This will result in generation of an incomplete XML file without any relations defined.

6.9.2 SchemaFactory

The static method createSchema in class dk.itu.bogm.db.db2xml.SchemaFactory in Appendix F maps all the information retrieved from the DatabaseMetadata object to objects of the classes depicted in Figure 4.18 on page 44.

Furthermore the SchemaFactory class will examine all Relations and determine whether there are many-to-many relations defined in the database. A many-to-many relation is identified if two tables have a many relation to the same table.

If the linking table in a Many2ManyRelation only has two columns (two foreign keys), the Table object is deleted from the Schema object, since this table should not be represented as an Entity Bean in the generated XML file, but as a many-to-many relation. Instead of the deleted table, relation objects are created between the two tables. If the linking table in a Many2ManyRelation contains more than two columns the table is kept to be mapped to an Entity Bean, and thus the Many2ManyRelation is deleted.

6.9.3 Type conversion — from SQL types to Java types

When information on each column in a table is retrieved the name and datatype of the column is stored. But the datatype needs to be stored as a Java type not as a SQL type. We use MiddleGen

class Sql2Java.java found in Appendix F to perform type conversions between SQL types and Java types.

```
private static void getColumns(...) throws SQLException{
...
   columnRs = databaseMetaData.getColumns(...);
   while (columnRs.next()) {
     int type = columnRs.getInt("DATA_TYPE");
     String dataType = Sql2Java.getFirstPreferredJdbcProperty(type);
     String columnName = columnRs.getString("COLUMN_NAME");
     //check if coloumn is a primary key.
     if(table.getPk().contains(columnName) & table.getPk().size() == 1){
       ....
        if(Sql2Java.isPrimitiveType(dataType))
          //this column is a PK, and cannot be a primitive type.
          dataType = dataType + "PK";
     }
     dataType = Sql2Java.getJavaTypeForJdbcProperty(dataType);
     table.addColumn(columnName, dataType);
   }
   ...
}
```

Figure 6.21: Code snippet from SchemaFactory.java

One thing the MiddleGen project's Sql2Java type converter does not check is that, according to the EJB2.0 specification a non-compound primary key may not be a primitive. As shown in the code snippet in Figure 6.21 there is now a built-in check to make sure that primary key columns cannot be mapped to a primitive type. If a column is a primary key and the datatype is primitive, a "PK" is appended to the datatype. When performing the call to the method getJavaTypeForJdbcProperty from the Sql2Java class, the datatype is mapped to the correct Java type, making sure that primary keys are not mapped to primitives. See Figure 6.22.

```
public static String getJavaTypeForJdbcProperty(String prop) {
        return (String)_propertyTypeMap.get(prop);
}
...
static { // modified by bogm
        _propertyTypeMap.put("byte", "byte");
        _propertyTypeMap.put("bytePK", "java.lang.Byte");
        _propertyTypeMap.put("short", "short");
        _propertyTypeMap.put("shortPK", "java.lang.Short");
        _propertyTypeMap.put("int", "int");
        _propertyTypeMap.put("intPK", "java.lang.Integer");
        _propertyTypeMap.put("long", "long");
        _propertyTypeMap.put("longPK", "java.lang.Long");
...
}
```

Figure 6.22: Code snippet showing our modifications of Sql2Java.java

Chapter 7

Test

A test is necessary to determine whether 3E works as intended. There are many ways to test a program — basically none of them can prove that a program contains no errors, but they can make it plausible that the program works as expected. Two techniques for making a systematic and effective test of a program are the structural test and the functional test. As structural and functional test are complementary, a full test might include both [21]. For time reasons this project will only include a functional test of 3E, while a structural test is left for future work.

The goal of a functional test is to make sure that 3E generates the expected EJB application. This can be done at different levels: One is to make a functional test of 3E itself, another is to make a functional test of the EJB application generated by 3E. Here, the functional test will be separated into three steps:

1. Does 3E perform the checks that we expect it to: For instance, is it checked that an Entity Bean does not have two fields with the same name or that a Relation does always refer to two existing Entity Beans?

2. Can 3E generate an implementation of the EJB application defined in an XML input file using all specific EJB features supported by 3E, and is the implementation correct with respect to the EJB2.0 specification?

3. Can the generated EJB application be deployed and run on the J2EE application servers supported by 3E?

The first step is done by creating a test data set for each of those checks that 3E claims to perform. The second and third step are joined together by creating a test case, which uses all possible EJB features supported by 3E. First 3E is used to generate the implementation of the test case and afterward the generated EJB application must prove capable of deploying and running on J2EE application servers.

7.1 Test of checks made by 3E

We expect 3E to perform checks on data in the application model before any code is generated. A test is made to convince ourselves that these checks are carried out.

A list of all the checks that 3E claims to perform can be found in Section 5.4.2 on page 56. A test data set is made for each of the checks pointed out in the list. Each test data set should cover common as well as rare input values and the outcome must be compared with what was expected. In appendix C all test data sets with input data and expected as well as observed output can be found. The test was made by providing 3E with an XML input file reflecting the specified input data. If, for instance, the input data is specified to be 'Package-name is empty String', an XML input file with the attribute `package-name` left as an empty String is given to 3E.

We are aware that the test is not exhaustive as some of the test data sets does not fully cover common as well as rare input values. Yet, all the checks have been tested with at least one invalid input value and one valid input value.

Two errors where found. First, a default-transaction-attribute for an Entity Bean should always exactly match 'Required', 'RequiresNew' or 'Mandatory'. It turned out that the check was made alright, but when it came out unsuccessful, a VisitorErrorException was thrown instead of the expected IllegalInputException (Remember the VisitorErrorException indicates an error in the implementation of 3E, while an IllegalInputException indicates that the developer gave some wrongful input).

Secondly, in a relation the multiplicity of a role should always be exactly 'One' or 'Many'. Again it turned out that the check was made alright, but a VisitorErrorException was thrown instead of an IllegalInputException.

Both errors where corrected and the test was made a second time where all test results came out as expected. The present thesis describes the corrected version of 3E.

7.2 Test case

A test case is created to cover all EJB features supported by 3E. It was considered to use a real life case, but it turned out to be quite difficult to find a suitable one. Instead the test case is inspired by the example used throughout [18] to explain and demonstrate the fundamentals of the EJB2.0 specification. By using this example to test 3E, the test will cover the fundamental concepts of the EJB2.0 specification.

The example is about the imaginary business Titan which is a cruise line company. A series of workbooks have been published in conjunction to [18] to provide readers with step-by-step instructions for running the Titan business examples on different J2EE application servers. Each workbook is designed for a specific server. As 3E currently supports the JBoss and the WebLogic6.1 J2EE application servers, the workbooks for these two servers [2, 20] have been the basis for the creation of our test case.

Originally Session Beans are used to represent some of the concepts of the Titan business. As 3E cannot generate Session Beans, the original Titan business is slightly modified. Figure 7.1 illustrates how the Titan business looks in the test case.

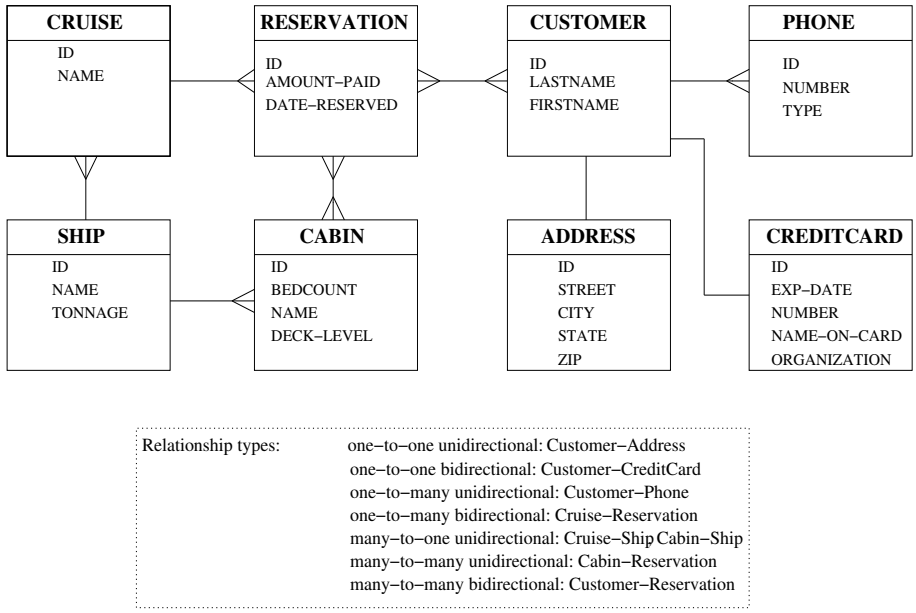

Figure 7.1: The Titan business

7.2.1 Creating the XML input file

First of all, the test case includes the creation of an XML input file that reflects the Titan business. Thus the XML input file defines Entity Beans for each of the participants in the Titan business: Cabin, Ship, Cruise, Reservation, Customer, Address, Phone and CreditCard. Fields and relations between Entity Beans are specified to match information implied in Figure 7.1. Furthermore, methods are added to the Entity Beans to match the examples in the workbooks. The final XML input file used in the Titan business test case on JBoss can be found in Appendix D and is also distributed with 3E (`TitanCruise.xml` located in the subdirectory `examples`). As we want to test 3E with regard to both the JBoss server and the WebLogic6.1 server, actually two XML input files were made. They should be identical with the exception of the specified type of server, but it turned out that also the implementation of a method in Customer accessing the Phone home interface had to differentiate depending on the type of server — this will be discussed in Chapter 8.

Note that the Titan business is created with the special intention to use all kinds of relationships and all kinds of methods specified in the EJB2.0 specification. By running 3E on the test case XML input file, the following is tested:

- Can 3E generate an implementation of an EJB2.0 Entity Bean with CMP fields corresponding to the definition given in the XML input file? The implementation must include an implementation of the Entity Beans primary key, whether this is declared to be automatically generated by 3E or not.
- Can 3E implement all kinds of methods specified in the EJB2.0 specification? I.e. create-methods, find-methods, select-methods, home-methods and business-methods.
- Can 3E implement all kinds of relationships specified in the EJB2.0 specification? I.e. all seven kinds created by combining one-to-one, one-to-many and many-to-many with the additional feature of a relationship being either unidirectional or bidirectional.

7.2.2 Testing the generated EJB application

The next step is to determine if the EJB application generated by 3E from the XML input file is a correct implementation. The generated EJB application consists of:

8 BeanClasses	CabinBean.java, ShipBean.java, CruiseBean.java, ReservationBean.java, CustomerBean.java, AddressBean.java, PhoneBean.java and CreditCardBean.java
8 LocalComponentInterfaces	Cabin.java, Ship.java, Cruise.java, Reservation.java, Customer.java, Address.java, Phone.java and CreditCard.java
8 LocalHomeInterfaces	CabinHome.java, ShipHome.java, CruiseHome.java, ReservationHome.java, CustomerHome.java, AddressHome.java, PhoneHome.java and CreditCardHome.java
Deployment descriptor	ejb-jar.xml
For WebLogic6.1	weblogic-cmp-rdbms-jar.xml and weblogic-ejb-jar.xml

Inspecting and compiling the EJB application

First, we can convince ourselves that the generated EJB application *seems* to be correct by inspecting and compiling the generated code.

An inspection requires that we determine whether all expected files are found, whether expected fields and methods for Entity Beans seems to be present and whether relationships among Entity Beans are represented. Note that expected files are a BeanClass, a LocalCompontInterface and a LocalHomeInterface for each Entity Bean together with the deployment descriptors for the specified type of server.

Compiling the code will check that the code is syntactically correct, all fields and variables are declared etc. The implementation of methods defined by the developer in the XML input file can of course contain syntax errors, but one cannot lay the blame on 3E for such errors. Note that compiling EJB2.0 code requires the class path to include some additional packages, for instance the Enterprise JavaBeans package `javax.ejb`.

By inspection all expected files are found and the content seems to be correct: An implementation of all fields, methods and relations defined in the XML input are present. Also, the generated code is compiled without any problems, meaning that the generated code is statically correct.

Deploying and running the EJB application

Secondly, the generated EJB application can be tested by deploying the EJB application on JBoss and WebLogic6.1 respectively. The deployed EJB application can be tested further by running it against clients. For this purpose 22 client JSP scripts have been developed. The client JSP pages are deployed on the J2EE application server together with the EJB application. The client JSP pages used in this test are copies of the client JSP pages used in the workbook examples, only slightly modified to match the EJB application generated by 3E. All the client JSP pages test a specific feature of the EJB application, i.e. the creation of EJBs using create methods, a specific

relationship, the use of set methods or business methods to modify relationships or the execution of a specific finder or select method (Some select methods causes home methods to be executed as well). In summary, the following is tested by the JSP clients:

- Execution of create methods
- A one-to-one bidirectional relationship (Customer-CreditCard)
- A one-to-one unidirectional relationship (Customer-Address)
- A one-to-many unidirectional relationship (Customer-Phone)
- A many-to-one unidirectional relationship (Cruise-Ship)
- A one-to-many bidirectional relationship, including the use of set methods and business methods respectively to modify the relation (Cruise-Reservation)
- A many-to-many bidirectional relationship, including the use of set methods and business methods respectively to modify the relation (Customer-Reservation)
- A many-to-many unidirectional relationship, including the use of set methods to modify the relation (Cabin-Reservation)
- Execution of simple find and select methods, including the use of the IN operator in queries
- Execution of more complex find methods

Appendix E on page 115 gives a short description of each of the 22 JSP clients together with a description of the expected and the actual outcome after executing each of the client scripts.

Two errors were found when deploying and running the EJB application on JBoss and WebLogic6.1 respectively:

1. In the weblogic-cmp-rdbms-jar.xml the mapping of relationships to database tables is specified. The mapping of a many-to-many relationship is done by creating a new table with two columns containing foreign keys to the involved Entity Beans. Each Role in the relationship must be represented by a foreign key pointing to the Entity Bean referred by the Role. For instance, the Customer-Reservation relationship should be represented by a new table having a column CUSTOMER_ID containing foreign keys to the CUSTOMER table and a column RESERVATION_ID containing foreign keys to the RESERVATION table. It turned out that 3E made a wrong mapping by making the column CUSTOMER_ID refer to the RESERVATION table and the column RESERVATION_ID refer to the CUSTOMER table.

2. In the file ejb-jar.xml each find or select method is defined within a <query> tag. The tag must always include a <method-params> tag, even when the method does not take any parameters. 3E only included the <method-params> tag if the method did take parameters.

Both errors were corrected and the EJB application was re-generated and tested a second time on both servers. The second test revealed no errors. The present thesis describes the corrected version of 3E.

7.2.3 Transactions and rollback

A valuable feature for EJB applications is the support of transactions. To challenge the EJB application generated by 3E, an additional JSP client was made whose only purpose is to provoke a rollback. The client JSP creates Cruise A, Cruise B and Reservation 1-2-3-4-5-6, linking Reservation 1-2-3 to Cruise A and Reservation 4-5-6 to Cruise B. Afterwards, within the same transaction, the names of both cruises are changed and all reservations originally linked to Cruise A are moved to Cruise B. Before the transaction ends, an exception is thrown. During and after the transaction, values returned by get methods are compared to values found in the database. As the transaction fails, no changes made during the transaction should appear in the database after the transaction has ended. This turned out to be true in our test as well.

7.2.4 Compound key

A special feature of 3E is the support of compound keys for Entity Beans. A sub-example of the test case was made to test this feature. The sub-example defines only Ship, Cruise and Reservation from the original test case. Both Cruise and Reservation have a compound primary key: For Cruise it consists of two Integers, for Reservation it consists of an Integer and a String. Three JSP clients test whether referential integrity is maintained during various modifications of relations between ships, cruises and reservations, similar to what was tested in the original test case (ex72_a.jsp, ex72_b.jsp and ex72_c.jsp in appendix E). No errors were found.

7.3 Conclusion on the test of 3E

The tests performed confirm that 3E does carry out the checks that it claims to and that the generated EJB application can deploy and run on JBoss and WebLogic6.1 J2EE application servers. The test of the checks made by 3E could be more exhaustive, but still it gives us confidence that the checks are performed. The Titan business test case is a solid test of whether an EJB application generated by 3E is correct, and the test results strongly indicates that 3E works as intended.

We are aware that a functional test of 3E could or should include more than described in this chapter. For instance, the XML schema describing the language for making a definition of an application could be checked with regard to the specification of data being optional or required. The line numbers found in debug messages should be tested to see that they do point back to the right place in the XML input file. Also, it could be interesting to time 3E while generating 10, 100 or 200 Entity Beans: We would expect the run-time to be O(n), while if not, it might indicate a performance bug in the implementation of 3E.

Chapter 8

Conclusion

This thesis has described the development of 3E, an open source command line tool for generating EJB applications. The purpose of our work was not solely to create a new tool, but also to gain experience with the development process and to improve our programming skills. In this final chapter we will try to evaluate both the product of our efforts, that is 3E, and the process of getting here.

8.1 3E

The purpose of the open source code generator 3E presented in this report is to save development time and spare developers the tedious work of writing EJB applications that conforms to the EJB2.0 specification. First of all, Writing EJB applications involves many restrictions and rules that are not all checked by the Javac compiler. E.g. should find methods be specified in the bean class or in the deployment descriptor and should they be exposed in the interfaces? Secondly, the work is repeatedly, since an Entity Bean must be described in the bean class as well as in the XML deployment descriptor. It is our opinion that 3E serves its purpose since the benefits of 3E are: Writing less code and ensuring correctness of the applications generated.

The architectural design of 3E is divided into four separate areas:

One is a language for expressing a specification of the EJB application to be generated (the application specification). This is done using an XML Schema describing how the specification can be expressed using XML.

Another is to parse the specification given in XML to a Java object structure being the internal representation of the EJB application to generate (the application model). This is done using the open source tool Castor XML, leaving us complete control of which XML elements to be parsed to which objects.

The third area is to check and complete the data in the application model. This functionality is implemented using the Visitor design pattern, traversing the application model several times to ensure its consistency and correctness.

The last area is to generate the Java and XML code needed for a complete EJB application. We chose the open source template language Velocity to perform this task.

One of the requirements for 3E was that 3E should be easy to maintain and extend with improved functionality. It is our believe that the architectural design of 3E supports this requirement by separating the various tasks. Furthermore the requirement is met partly by choosing the

XML parser tool Castor XML, letting changes in the application model or in the XML Schema be easily adjusted, and partly by choosing the template language Velocity, which supports the separation of the application model from the generating code. Also, Velocity makes it very easy to create new templates to generate other files. Still, even though Velocity has been a very easy language to master, there is one single drawback to mention: Since the formatting of Velocity functions in the templates needs to be left justified to generated nicely formatted code, the Velocity templates become harder to read. E.g. it is very hard to tell if a '#end' is ending either a loop or a conditional statement, especially if they are mixed together. Still, we find the benefits from using Velocity high enough to accept this drawback.

The final version of 3E has a complete implementation of the following features which are identical to the most highly prioritized requirements described in Section 3.2 by an interview of Morten Madsen, an EJB developer working for the software company TietoEnator:

- Generation of an XML input file for 3E reflecting tables and relations of a running database.
- Generation of container managed persistence Entity Beans with component and home interfaces that conforms to the EJB2.0 specification. Including an XML deployment descriptor and server specific deployment descriptors for the WebLogic6.1 server.
- The ability to express a methods transaction level.
- The ability to express the 7 kinds of Entity Bean relationships.
- Support for generating globally unique primary keys automatically.
- Producing debug messages in the generated code that points back to the input file.

8.1.1 Discussion and future work

Morten Madsen evaluated 3E in the final stage of the development process. Not only was he impressed by the number of features supported, he was also fond of how easy writing the XML input file was. Morten Madsen even stated that he would consider using 3E in his work, provided that it was extended to support the Oracle application server as well. Extending 3E to support the Oracle application server is not a complicated job. All that needs to be done is write a new Velocity template reflecting the specification for Oracle's server specific deployment descriptor.

The performed test of 3E did not reveal many errors. Those that were found have been corrected in the distributed version of 3E. A more thorough test with an extended amount of various input data might reveal more errors. Still, the test convince us that 3E does generate EJB applications conforming to the EJB2.0 specification.

One of the requirements for 3E was that the tool should generate EJB applications that are portable across various J2EE application servers. However, the test of 3E revealed that this is not possible when implementing a method in an Entity Bean that looks up the home interface for another Entity Bean, as the way to perform this lookup differentiates with the type of server. Thus, on Jboss the lookup is performed by jndiEnc.lookup("local/PhoneEJB") while on WebLogic6.1 the lookup is performed by jndiEnc.lookup("PhomeHomeLocal"). Thus two different XML input files with slightly changed implementation of this method was needed to generate deployable EJB applications for each server. However, this is not an error in 3E, but reveals a weakness in the J2EE specification not stating one specific way to perform a lookup in a J2EE

application server.

Thus, 3E has proved to be a useful tool, even though not the complete list of requirements mentioned in Chapter 3 is implemented. Also, when testing 3E Morten Madsen had several more ideas for additional features. These ideas, our own suggestions for improvements and original requirements not yet implemented are listed here as inspiration for future work.

- Implementation of a configuration class where lists of all legal values are collected. E.g. the class would hold a list of all legal transaction attributes, a list of all servers and a list of all Java types that can be automatically generated for primary keys. This class would be easier to maintain than the current implementation where these values are spread out through the visitor classes.
- Extending 3E to perform new tasks such as generating Javadoc.
- The possibility to specify a more flexible package structure. At the moment all generated java files are included in the same package.
- Currently a full class name must be specified in the XML input file when specifying parameters for find and select methods and when specifying cmp-pk-fields. It would be an improvement if only the type was to be specified and 3E could subsequently retrieve the full class name.
- Improving the error messages to use the same functionality as the debug messages, and thus point back to the exact line in the input file that caused the error to occur.
- Implementation of support of cascade-delete in one-to-one and one-to-many relationships. The cascade-delete is a functionality to control referential integrity when removing an EJB instance.
- Provide restraints to the method classes that should not throw application specific exceptions. Currently there is no restraint, so it is possible to specify an application specific exception to a method which may not throw this type of exception. The only constraint to prevent this from happening is specified in the ejbScema4_0.xsd.
- Currently there is no syntax checking of the Java code specified in the method-body elements in the XML input file. This Java code is only checked when the generated code is compiled. A feature checking correct syntax of the Java code could be a future improvement.
- Generation of value objects for the Entity Beans. This is one of the original requirements though not highly prioritized by Morten Madsen.
- Generation of JSP clients. Also one of the original requirements not considered to be very important by Morten Madsen.

8.2 The development process

We have learned much about the development process from this project.

Throughout the project we have chosen open source tools to accomplish different goals, both in the implementation of 3E and to support the general development process. Our experience is

that open source tools offer more freedom and are more flexible than commercial tools. Unfortunately many open source tools also have poor documentation and useless guides for installation and use (this is also true for some commercial products though). We find that this experience is reflected in 3E, why 3E is an open source tool with good documentation and a user manual covering both installation and use of the tool.

In the development process we have gained experience with different tools almost indispensable in a project of this size. Tortoise CVS has been very useful in the process of working on the same files without worrying about overwriting changes made by the other part. Ant made comprehensive and tiresome every day tasks easy. In future projects we will without question be advocates of using such tool to ease the development process.

With regard to our programming skills we are sure that they have improved and that they can become even better. Hopefully we will be involved in other projects in time to come to make such progress possible. We are still new in the field of software development, but have gained enough experience from this project to throw ourselves into future projects with our heads up.

Bibliography

[1] Apache. *Hompage of Ant*. URL: http://ant.apache.org/, January 2003.

[2] Bill Burke and Sacha Labourey. *JBoss 3.0 Workbook for EnterPrise JavaBeans*. Titan Books, Inc., 3rd edition, beta copy edition, 2002. Free download in pdf-format including examples available from URL: http://www.monson-haefel.com/titanbooks/.

[3] *Castor*. URL: http://castor.exolab.org/index.html. January 2003.

[4] *Castor XML Mapping*. URL: http://castor.exolab.org/xml-mapping.html. January 2003.

[5] J Craig Cleaveland. *Program generators with Java and XML*. Prentice-Hall, 2001. ISBN: 0-13-025878-4.

[6] *Homepage of Eclipse*. URL: http://www.eclipse.org, January 2003.

[7] *Homepage of EJBBuilder*. URL: http://ejbbuilder.sourceforge.net/, January 2003.

[8] *Homepage of EJBDoclet*. URL: http://sourceforge.net/projects/ejbdoclet, January 2003.

[9] *Homepage of EJBGen*. URL: http://ejbgen.sourceforge.net/, January 2003.

[10] Ralph Johnson & John Vlissides Erich Gamma, Richard Helm. *Design Patterns*. Addison-Wesley, 1995. ISBN: 0201633612.

[11] Robert Hustead. *Mapping XML to Java, Part 1*. URL: http://www.javaworld.com/javaworld/jw-08-2000/jw-0804-sax_p.html. Part of the JavaWorld.com Archive, and available only through a subscription.

[12] Ulrich W. Eisenecker Krzysztof Czarnecki. *Generative programming : Methods, tools, and applications*. Addison-Wesley, 2000. ISBN: 0-201-30977-7.

[13] Floyd Marinescu. *EJB design patterns: Advanced patterns, processes and idioms*. John Wiley and sons, 2001. ISBN:.

[14] Elliotte Rusty Harold & W. Scott Means. *XML in a Nutshell*. O'Reilly & Associates, Inc., 2001. A Desktop Quick Reference.

[15] Sun Microsystems. *Enterprise JavaBeans specification, version 2.0*. PDF download at URL: http://java.sun.com/products/ejb/docs.html, August 2001. version 2.0, Final release.

[16] Sun Microsystems. *Java API*. URL: `http://java.sun.com/j2se/1.3/docs/api/`, January 2003.

[17] *Homepage of MiddleGen*. URL: `http://boss.bekk.no/boss/middlegen/index.html`, January 2003.

[18] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, third edition edition, 2001. ISBN: 0-596-00226-2.

[19] Nana Dencker Bargisen & Anne Mette Mørkbak. Specialeforberedende projekt. Technical report, IT-C, Copenhagen DK, 2002.

[20] Greg Nyberg. *WebLogic Server 6.1 Workbook for EnterPrise JavaBeans*. Titan Books, Inc., 3rd edition edition, 2002. Free download in pdf-format including examples available from URL: `http://www.monson-haefel.com/titanbooks/`.

[21] Department of Mathematics Peter Sestoft, Royal Veterinary Physics, and Denmark Agricultural University. Systematic software test. English version 1, 1998-07-31.

[22] Roger S. Pressman. *Software engineering - A practitioner's approach*. R.S. Pressman & Associates, 5. edition edition, 2000. ISBN: 0 07 709677 0.

[23] Anders Møller & Michael I. Schwartzbach. *The XML Revolution, Technologies for the future Web*. URL: `http://www.brics.dk/~amoeller/XML/index.html`, March 2000. Latest revision: October 2002.

[24] *Homepage of srcGen*. URL: `http://sourceforge.net/projects/srcgen/`, January 2003.

[25] Velocity Documentation Team. *Velocity*. URL: `http://jakarta.apache.org/velocity/index.html`. January 2003.

[26] *Homepage of Tortoies CVS*. URL: `http://www.tortoisecvs.org`, January 2003.

[27] W3Schools.com. *Introduction to XML Schema*. URL: `http://www.w3schools.com/schema/schema_intro.asp`.

[28] *Homepage of XDoclets*. URL: `http://xdoclet.sourceforge.net/`, January 2003.

Appendix A

Generating Java code and XML from JSP

From the JSP-file shown in figure A.1 the Java code shown in figure A.2 can be generated. For this to be done the program needs to define a Tag Library *code* implementing the following tags: *ReadFileTag*, *ClassNameTag*, *FieldTypeTag* and *FieldNameTag*. As a reference to the application model is obtained, each tag can traverse the application model and read the missing information to generate the Java code.

Figure A.3 shows the XML generated by the JSP file shown in Figure 4.11 on page 37.

```
<%@taglib uri="/WEB-INF/JavaCodeGeneration.tld" prefix="code" %>

<PRE>

<code:read-file>
public class <code:class-name/>{
  private <code:field-type/> <code:field-name/>;

  public <code:class-name/>(<code:field-type/> <code:field-name/>){
    set<code:field-name/>(<code:field-name/>);
  }

  public void set<code:field-name/>(<code:field-type/> <code:field-name/>){
    this.<code:field-name/> = <code:field-name/>;
  }

  public <code:field-type/> get<code:field-name/>(){
    return this.<code:field-name/>;
  }
}
</code:read-file>

</PRE>
```

Figure A.1: JSP file generating Java

```
public class Person{
  private java.lang.String name;

  public Person(java.lang.String name){
    setname(name);
  }

  public void setname(java.lang.String name){
    this.name = name;
  }

  public java.lang.String getname(){
    return this.name;
  }
}
```

Figure A.2: Java code generated from JSP

```
<books>
   <book isbn="123">
      <title>Web Servers for Fun and Profit</title>
      <quantity>10</quantity>
      <price>$17.95</price>
   </book>
   <book isbn="456">
      <title>Web Programming for Mobile Devices</title>
      <quantity>2</quantity>
      <price>$38.95</price>
   </book>
   <book isbn="789">
      <title>Duke: A Biography of the Java Evangelist</title>
      <quantity>25</quantity>
      <price>$15.00</price>
   </book>
</books>
```

Figure A.3: XML generated by the JSP file in Figure 4.11

Appendix B

Files generated by the input file Simple.xml

From the Simple.xml input file shown i Figure 5.1 on page 5.1 the following files are generated:

Address.java the local component interface for the Address EJB, see page 93

AddressHome.java the local home interface for the Address EJB, see page 93

AddressBean.java the bean class for the Address EJB, see page 94

AddressPK.java the primary key class for the Address EJB, see page 95

Customer.java the local component interface for the Customer EJB, see page 96

CustomerHome.java the local home interface for the Customer EJB, see page 96

CustomerBean.java the bean class for the Customer EJB, see page 97

GUIDProvider.java the GUID provider class, see page 98

ejb-jar.xml the deployment descriptor, see page 100

B.1 Address.java

```
/*********************************************************************
FILE: Address.java
DESCRIPTION: The local component interface for the Address EJB
*********************************************************************/

package dk.itu.bogm;

import javax.ejb.*;

public interface Address extends EJBLocalObject{

   // GET'ers
   public java.lang.Long getPk1();
   public java.lang.String getPk2();
   public java.lang.String getAddress();

   // SET'ers
   public void setAddress(java.lang.String address );

   // BUSINESS-methods
}
```

B.2 AddressHome.java

```
/*********************************************************************
FILE: AddressHome.java
DESCRIPTION: The local home interface for the Address EJB
*********************************************************************/

package dk.itu.bogm;

import javax.ejb.*;

public interface AddressHome extends EJBLocalHome{

   // CREATE-methods
   public dk.itu.bogm.Address createAddress() throws  CreateException ;

   // HOME-methods

   // FIND-methods
   public Address findByPrimaryKey( AddressPK pk) throws  FinderException ;
}
```

B.3 AddressBean.java

```
/*********************************************************************
FILE: AddressBean.java
DESCRIPTION: the bean class for the Address EJB
*********************************************************************/

package dk.itu.bogm;

import javax.ejb.*;

public abstract class AddressBean implements EntityBean{

   // GET'ers
   abstract public java.lang.Long getPk1();
   abstract public java.lang.String getPk2();
   abstract public java.lang.String getAddress();

   // SET'ers
   abstract public void setPk1(java.lang.Long pk1);
   abstract public void setPk2(java.lang.String pk2);
   abstract public void setAddress(java.lang.String address);

   // BUSINESS-methods

   // HOME-methods

   // PRIVATE-methods

   // SELECT-methods

   // ejbCREATE-methods
   public AddressPK ejbCreateAddress ( ) throws CreateException {
      this.setPk1(new Long(new java.security.SecureRandom().nextLong()));
      this.setPk2(GUIDProvider.createGUIDProvider().getGUID());
      return null;
   }

   // ejbPostCREATE-methods
   public void ejbPostCreateAddress() throws  CreateException {
   }

   // STANDARD-code
   private EntityContext entityContext;

   public void setEntityContext(EntityContext entityContext){
      this.entityContext = entityContext;
   }

   public EntityContext getEntityContext(){
      return this.entityContext;
   }

   public void unsetEntityContext() {
      this.entityContext = null;
   }

   public void ejbActivate(){ }
```

```
   public void ejbPassivate(){ }

   public void ejbLoad(){ }

   public void ejbStore(){ }

   public void ejbRemove() throws RemoveException{ }
}
```

B.4 AddressPK.java

```
/*********************************************************************
FILE: AddressPK.java
DESCRIPTION: the primary key class for the Address EJB
*********************************************************************/

package dk.itu.bogm;

public class AddressPK implements java.io.Serializable{
   public java.lang.Long pk1;
   public java.lang.String pk2;

   //CONSTRUCTORS
   public AddressPK(){}

   public AddressPK( java.lang.Long pk1, java.lang.String pk2){
      this.pk1 = pk1;
      this.pk2 = pk2;
   }

   // GET'ers
   public java.lang.Long getPk1(){
      return this.pk1;
   }

   public java.lang.String getPk2(){
      return this.pk2;
   }

   public boolean equals(Object obj){
      if(obj == null)
         return false;
      if(this == obj)
         return true;
      if(this.getClass() != obj.getClass())
         return false;
      AddressPK other = (AddressPK)obj;
      if(!(this.pk1.equals( other.pk1 )))
         return false;
      if(!(this.pk2.equals( other.pk2 )))
         return false;
      else
         return true;
   }

   public int hashCode(){
```

```
      int hashCode = 0;
      hashCode = hashCode^pk1.hashCode();
      hashCode = hashCode^pk2.hashCode();
      return hashCode;
   }
}
```

B.5 Customer.java

```
/*******************************************************************
FILE: Customer.java
DESCRIPTION: the local component interface for the Customer EJB
*******************************************************************/

package dk.itu.bogm;

import javax.ejb.*;

public interface Customer extends EJBLocalObject{

   // GET'ers
   public java.lang.String getPk();
   public java.lang.String getFirstName();
   public java.lang.String getLastName();
   public Address getHomeAddress();

   // SET'ers
   public void setFirstName(java.lang.String firstName );
   public void setLastName(java.lang.String lastName );
   public void setHomeAddress(Address homeAddress );

   // BUSINESS-methods
}
```

B.6 CustomerHome.java

```
/*******************************************************************
FILE: CustomerHome.java
DESCRIPTION: the local home interface for the Customer EJB
*******************************************************************/

package dk.itu.bogm;

import javax.ejb.*;

public interface CustomerHome extends EJBLocalHome{

   // CREATE-methods
   public dk.itu.bogm.Customer createCustomer() throws CreateException ;

   // HOME-methods

   // FIND-methods
   public Customer findByPrimaryKey( java.lang.String pk) throws FinderException ;
}
```

B.7 CustomerBean.java

```
/*******************************************************************
FILE: CustomerBean.java
DESCRIPTION: the bean class for the Customer EJB
*******************************************************************/

package dk.itu.bogm;

import javax.ejb.*;

public abstract class CustomerBean implements EntityBean{

   // GET'ers
   abstract public java.lang.String getPk();
   abstract public java.lang.String getFirstName();
   abstract public java.lang.String getLastName();
   abstract public Address getHomeAddress();

   // SET'ers
   abstract public void setPk(java.lang.String pk);
   abstract public void setFirstName(java.lang.String firstName);
   abstract public void setLastName(java.lang.String lastName);
   abstract public void setHomeAddress(Address homeAddress);

   // BUSINESS-methods

   // HOME-methods

   // PRIVATE-methods

   // SELECT-methods

   // ejbCREATE-methods
   public java.lang.String ejbCreateCustomer ( ) throws  CreateException {
      this.setPk(GUIDProvider.createGUIDProvider().getGUID());
      return null;
   }

   // ejbPostCREATE-methods
   public void ejbPostCreateCustomer() throws  CreateException {
   }

   // STANDARD-code
   private EntityContext entityContext;

   public void setEntityContext(EntityContext entityContext){
      this.entityContext = entityContext;
   }

   public EntityContext getEntityContext(){
      return this.entityContext;
   }

   public void unsetEntityContext() {
      this.entityContext = null;
   }
```

```
   public void ejbActivate(){ }

   public void ejbPassivate(){ }

   public void ejbLoad(){ }

   public void ejbStore(){ }

   public void ejbRemove() throws RemoveException{ }
}
```

B.8 GUIDProvider.java

```
/*********************************************************************
FILE: GUIDProvider.java
DESCRIPTION: the GUID provider class
*********************************************************************/

package dk.itu.bogm;

import java.security.*;
import java.net.*;

public class GUIDProvider{
   private static SecureRandom seeder;
   private static GUIDProvider guid;
   private static String midValue;

   //private constructor implementing Singleton pattern.
   private GUIDProvider(){
      try{
         //get internet address
         InetAddress inet = InetAddress.getLocalHost();
         byte [] bytes = inet.getAddress();
         String hexInetAddress = hexFormat(getInt(bytes), 8);
         //get hashCode for this object
         String thisHashCode = hexFormat(System.identityHashCode(this), 8);
         // set the midValue
         midValue = hexInetAddress + thisHashCode;
         seeder = new SecureRandom();
      }
      catch(Exception e){
         System.out.println("Error while creating GUIDProvider: " + e);
      }
   }

   public static GUIDProvider createGUIDProvider(){
      if(guid == null){
         guid = new GUIDProvider();
         return guid;
      }
      else
         return guid;
   }

   public String getGUID(){
      //get current time
```

```
      long timeNow = System.currentTimeMillis();
      //get int value as unsigned
      int timeLow = (int)timeNow & 0xFFFFFFFF;
      //get nest random value
      int node = seeder.nextInt();
      return (hexFormat(timeLow, 8) + midValue + hexFormat(node, 8));
   }

   // The following three methods are taken from Middlegen
   // and can be found at:
   // http://boss.bekk.no/boss/middlegen/samples/airline/
   // interfaces/SequenceUtil.java.html
   private static int getInt(byte bytes[]) {
      int i = 0;
      int j = 24;
      for (int k = 0; j >= 0; k++) {
         int l = bytes[k] & 0xff;
         i += l << j;
         j -= 8;
      }
      return i;
   }

   private static String hexFormat(int i, int j) {
      String s = Integer.toHexString(i);
      return padHex(s, j) + s;
   }

   private static String padHex(String s, int i) {
      StringBuffer tmpBuffer = new StringBuffer();
      if (s.length() < i) {
         for (int j = 0; j < i - s.length(); j++) {
            tmpBuffer.append('0');
         }
      }
      return tmpBuffer.toString();
   }
}
```

B.9 ejb-jar.xml

/**
FILE ejb-jar.xml
DESCRIPTION: the deployment descriptor
**/

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC
"-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>

   <enterprise-beans>

      <entity>
         <ejb-name>CustomerEJB</ejb-name>
         <local-home>dk.itu.bogm.CustomerHome</local-home>
         <local>dk.itu.bogm.Customer</local>
         <ejb-class>dk.itu.bogm.CustomerBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.String</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-version>2.x</cmp-version>
         <abstract-schema-name>Customer</abstract-schema-name>
         <cmp-field>
            <field-name>pk</field-name>
         </cmp-field>
         <cmp-field>
            <field-name>firstName</field-name>
         </cmp-field>
         <cmp-field>
            <field-name>lastName</field-name>
         </cmp-field>
         <primkey-field>pk</primkey-field>
      </entity>

      <entity>
         <ejb-name>AddressEJB</ejb-name>
         <local-home>dk.itu.bogm.AddressHome</local-home>
         <local>dk.itu.bogm.Address</local>
         <ejb-class>dk.itu.bogm.AddressBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>dk.itu.bogm.AddressPK</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-version>2.x</cmp-version>
         <abstract-schema-name>Address</abstract-schema-name>
         <cmp-field>
            <field-name>pk1</field-name>
         </cmp-field>
         <cmp-field>
            <field-name>pk2</field-name>
         </cmp-field>
         <cmp-field>
            <field-name>address</field-name>
         </cmp-field>
```

```
      </entity>

   </enterprise-beans>

   <relationships>

      <ejb-relation>
         <ejb-relation-name>Customer-Address</ejb-relation-name>

         <ejb-relationship-role>
            <ejb-relationship-role-name>
               Customer-has-address
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
               <ejb-name>CustomerEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
               <cmr-field-name>homeAddress</cmr-field-name>
            </cmr-field>
         </ejb-relationship-role>

         <ejb-relationship-role>
            <ejb-relationship-role-name>
               Address-belongs-to-customer
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
               <ejb-name>AddressEJB</ejb-name>
            </relationship-role-source>
         </ejb-relationship-role>
      </ejb-relation>

   </relationships>

   <assembly-descriptor>

      <container-transaction>
         <method>
            <ejb-name>CustomerEJB</ejb-name>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Required</trans-attribute>
      </container-transaction>

      <container-transaction>
         <method>
            <ejb-name>AddressEJB</ejb-name>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Required</trans-attribute>
      </container-transaction>

   </assembly-descriptor>

</ejb-jar>
```

Appendix C

Test of checks done by visitors in 3E

Test: Data required from developer is present

Input data	Expected result	Test result
Missing packagename	IllegalInputException: 'packagename in Model must not be null'	As expected
Missing deployment	IllegalInputException: 'deployment in Model must not be null'	As expected
Missing name of Entity Bean	IllegalInputException: 'name in EntityEjb must not be null'	As expected
Missing name of relation	IllegalInputException: 'name in Relation must not be null'	As expected
Missing type of server in deployment	IllegalInputException: 'typeOfServer in Deployment must not be null'	As expected
Missing source in role	IllegalInputException: 'source in Role must not be null'	As expected
Missing name of role	IllegalInputException: 'name in Role must not be null'	As expected
Missing multiplicity of role	IllegalInputException: 'multiplicity in Role must not be null'	As expected

...continues on next page...

Missing type of primary key field	IllegalInputException: 'type in CmpPkField must not be null'	As expected
Missing name of primary key field	IllegalInputException: 'name in CmpPkField must not be null'	As expected
Missing type of CMP field	IllegalInputException: 'type in CmpField must not be null'	As expected
Missing name of CMP field	IllegalInputException: 'name in CmpField must not be null'	As expected
Missing name of select method	IllegalInputException: 'name in SelectMethod must not be null'	As expected
Missing return-type of select method	IllegalInputException: 'returnType in SelectMethod must not be null'	As expected
Missing query of select method	IllegalInputException: 'query in SelectMethod must not be null'	As expected
Missing type of parameter	IllegalInputException: 'type in Parameter must not be null'	As expected
Missing name of parameter	IllegalInputException: 'name in Parameter must not be null'	As expected
Entity Bean Reservation has business method getName, where body of methodbody is missing	IllegalInputException: 'body in method getName in Reservation must not be null'	As expected
Entity Bean Ship has find method findByTonnage, where body of query is missing	IllegalInputException: 'body in query findByTonnage in Ship must not be null'	As expected

Test: String values required from developer is not an empty String

Input data	Expected result	Test result
Package-name is empty String	IllegalInputException: 'packageName in Model must not be an empty String'	As expected
Name of Entity Bean is empty String	IllegalInputException: 'name in EntityEjb must not be an empty String'	As expected

...continues on next page...

Name of relation is empty String	IllegalInputException: 'name in Relation must not be an empty String'	As expected
Type-of-server is empty String	IllegalInputException: 'typeOfServer in Deployment must not be an empty String'	As expected
Source in role is empty String	IllegalInputException: 'source in Role must not be an empty String'	As expected
Name of role is empty String	IllegalInputException: 'name in Role must not be an empty String'	As expected
Multiplicity of role is empty String	IllegalInputException: 'multiplicity in Role must not be an empty String'	As expected
Type of primary key field is empty String	IllegalInputException: 'type in CmpPkField must not be an empty String'	As expected
Name of primary key field is empty String	IllegalInputException: 'name in CmpPkField must not be an empty String'	As expected
Type of CMP field is empty String	IllegalInputException: 'type in CmpField must not be an empty String'	As expected
Name of CMP field is empty String	IllegalInputException: 'name in CmpField must not be an empty String'	As expected
Name of CMR field is empty String	IllegalInputException: 'name in CmrField must not be an empty String'	As expected
Name of select method is empty String	IllegalInputException: 'name in SelectMethod must not be an empty String'	As expected
Return-type of select method is empty String	IllegalInputException: 'returnType in SelectMethod must not be an empty String'	As expected
Field to be instantiated by create method is empty String	IllegalInputException: 'fieldToInstantiate in CreateMethod must not be an empty String'	As expected
Type of parameter is empty String	IllegalInputException: 'type in Parameter must not be an empty String'	As expected
Name of parameter is empty String	IllegalInputException: 'name in Parameter must not be an empty String'	As expected

...continues on next page...

Entity Bean Reservation has business method getName, where body of method-body is an empty String	'body in method getName in Reservation must not be an empty String'	As expected
Entity Bean Ship has find method findBy-Tonnage, where body of query is an empty String	'body in query findByTonnage in Ship must not be an empty String'	As expected

Test: At least one Entity Bean is specified

Input data	Expected result	Test result
No Entity Beans are specified	IllegalInputException: 'Model does not contain any Entity-ejbs. At least one should be specified'	As expected
Exactly one Entity Beans is specified	No errors	As expected
More than one Entity Beans are specified	No errors	As expected

Test: All Entity Beans have a primary key

Input data	Expected result	Test result
An Entity Bean Cruise with no primary key is specified	IllegalInputException: 'At least one cmpPkField must be spcified in EntityEjb Cruise'	As expected
Two Entity Beans Cruise and Ship with no primary keys are specified. Cruise is specified before Ship	IllegalInputException: 'At least one cmpPkField must be spcified in EntityEjb Cruise'	As expected
All specified Entity Beans have a primary key	No errors	As expected

Test: All specified transaction-attributes are valid

Input data	Expected result	Test result
The default-transaction-attribute for an Entity Bean Cruise is specified to be 'Requires'	IllegalInputException: 'DefaultTransactionAttribute not valid in EntityEjb Cruise>'	Gave VisitorErrorException
The default-transaction-attribute for an Entity Bean Cruise is specified to be 'required'	IllegalInputException: 'DefaultTransactionAttribute not valid in EntityEjb Cruise>'	Gave VisitorErrorException

...continues on next page...

All specified transaction-attributes are specified to be 'Required', 'RequiresNew' or 'Mandatory'	No errors	As expected

Test: The type of server is always 'JBoss' or 'WebLogice6.1'

Input data	Expected result	Test result
The type of server is specified be 'Bea'	IllegalInputException: 'TypeOfServer must match 'jBoss' or 'webLogic6.1' '	As expected
The type of server is specified be 'web-logic61'	IllegalInputException: 'TypeOfServer must match 'jBoss' or 'webLogic6.1' '	As expected
Type of server specified to be 'Jboss'	No errors	As expected
Type of server specified to be 'jboss'	No errors	As expected
Type of server specified to be 'Web-Logic6.1'	No errors	As expected

Test: The multiplicity of each role in relation must be exactly 'One' or 'Many'

Input data	Expected result	Test result
Multiplicity for role cruise-belongs-to-ship in relation Cruise_Ship is specified to be 'two'	IllegalInputException: 'Multiplicity for Role cruise-belongs-to-ship in relation Cruise_Ship MUST be exactly 'One' or 'Many'. Is two'	Gave Visitor-ErrorException
Multiplicity for role cruise-belongs-to-ship in relation Cruise_Ship is specified to be 'one'	IllegalInputException: 'Multiplicity for Role cruise-belongs-to-ship in relation Cruise_Ship MUST be exactly 'One' or 'Many'. Is one'	Gave Visitor-ErrorException
Multiplicity for role cruise-belongs-to-ship in relation Cruise_Ship is specified to be 'many'	IllegalInputException: 'Multiplicity for Role cruise-belongs-to-ship in relation Cruise_Ship MUST be exactly 'One' or 'Many'. Is many'	Gave Visitor-ErrorException
Multiplicity for role specified to be 'One'	No errors	As expected
Multiplicity for role specified to be 'Many'	No errors	As expected

Test: Name of select methods must start with 'ejbSelect'

Input data	Expected result	Test result
Name of select method specified to be 'select'	IllegalInputException: 'Name of SelectMethod MUST start with 'ejbSelect' '	As expected
Name of select method specified to be 'EjbSelect'	IllegalInputException: 'Name of SelectMethod MUST start with 'ejbSelect' '	As expected
Names of all select methods start with 'ejbSelect'	No errors	As expected

Test: Name of find methods must start with 'find'

Input data	Expected result	Test result
Name of find method specified to be 'get'	IllegalInputException: 'Name of FindMethod MUST start with 'find' '	As expected
Name of find method specified to be 'Find'	IllegalInputException: 'Name of FindMethod MUST start with 'find' '	As expected
Names of all find methods start with 'find'	No errors	As expected

Test: A find method named 'findByPrimaryKey' must not be specified by the developer

Input data	Expected result	Test result
A find method named 'findByPrimaryKey' is specified	IllegalInputException: 'FindMethod 'findByPrimaryKey' may not be given explicitly'	As expected
No find method named 'findByPrimaryKey' is specified	No errors	As expected

Test: Name of home method must not start with 'ejbHome', 'create', 'find' or 'remove'

Input data	Expected result	Test result
Name of home method specified to be 'ejbHomeGetAll'	IllegalInputException: 'Name of HomeMethod must NOT start with 'ejbHome', 'create', 'find' or 'remove' '	As expected
Name of home method specified to be 'createHome'	IllegalInputException: 'Name of HomeMethod must NOT start with 'ejbHome', 'create', 'find' or 'remove' '	As expected
Name of home method specified to be 'findAllCabins'	IllegalInputException: 'Name of HomeMethod must NOT start with 'ejbHome', 'create', 'find' or 'remove' '	As expected
Name of home method specified to be 'removeAllCabins'	IllegalInputException: 'Name of HomeMethod must NOT start with 'ejbHome', 'create', 'find' or 'remove' '	As expected
No home method is specified with a name starting with 'ejbHome', 'create', 'find' or 'remove'	No errors	As expected

Test: Name of business method must not start with 'ejb'

Input data	Expected result	Test result
A business method named 'ejbGetName' is specified	IllegalInputException: 'Name of BusinessMethod must NOT start with 'ejb' '	As expected
No business method is specified with a name starting with 'ejb'	No errors	As expected

Test: Name of create method must always start wiht 'create'

Input data	Expected result	Test result
A create method named 'makeShip' is specified	IllegalInputException: 'Name of CreateMethod MUST start with 'create' '	As expected
All create methods are specified with a name starting with 'create'	No errors	As expected

Test: A model must not contain two Entity Beans with the same name

Input data	Expected result	Test result
Two Entity Beans with name 'Cabin' are specified	IllegalInputException: 'Model has two EntityEjbs with name Cabin'	As expected
Three Entity Beans with name 'Cabin' are specified	IllegalInputException: 'Model has two EntityEjbs with name Cabin'	As expected
All specified Entity Beans have a unique name	No errors	As expected

Test: A model must not contain two relations with the same name

Input data	Expected result	Test result
Two relations wiht name 'Cruise_Ship' are specified	IllegalInputException: 'Model has two relations with name Cruise_Ship'	As expected
All specified relations have a unique name	No errors	As expected

Test: If primary key is not a compund key, the primary key type must not be a primitive type

Input data	Expected result	Test result
An Entity Bean Cruise with single primary key of type 'int' is specified	IllegalInputException: In Cruise: When primary key is not compound key, the CmpPkfield must not be primitive type	As expected
None of specified Entity Beans have primary keys of primitive type, unless the cmp-pk-field is part of a compound key	No errors	As expected

Test: A relation must always have exactly two roles

Input data	Expected result	Test result
A relation Cruise_Ship wiht no roles is specified	IllegalInputException: 'A relation must have exactly two roles. Cruise_Ship has 0 role(s)'	As expected
A relation Cruise_Ship wiht one role is specified	IllegalInputException: 'A relation must have exactly two roles. Cruise_Ship has 1 role(s)'	As expected
A relation Cruise_Ship wiht three roles is specified	IllegalInputException: 'A relation must have exactly two roles. Cruise_Ship has 3 role(s)'	As expected
All relations have two roles	No errors	As expected

Test: The roles of a relation must have unique names within the relation

Input data	Expected result	Test result
A relation Cabin_Ship wiht two roles named 'cabin-ship' is specified	IllegalInputException: 'Each role in a relation must have a unique name. In relation Cabin_Ship both roles have name cabin-ship'	As expected
All roles in relations have unique names within each relation	No errors	As expected

Test: A role must always refer to an existing Entity Bean

Input data	Expected result	Test result
A role 'cabin-belong-to-ship' in relation Cabin_Ship refers to an Entity Bean called Cabin. No Entity Bean called 'Cabin' exists	IllegalInputException: 'Cabin in relation Cabin_Ship is not known. No such EntityEjb exists'	As expected
All roles refers to existing Entity Beans	No errors	As expected

Test: If type of server is 'webLogic6.1', a dataSourceJndiName must be specified

Input data	Expected result	Test result
Type of server is 'webLogic6.1'. No dataSourceJndiName is specified	IllegalInputException: 'dataSourceJndiName in Model must not be null'	As expected
Type of server is 'webLogic6.1'. A dataSourceJndiName is specified, but it is an empty Strnig	IllegalInputException: 'dataSourceJndiName in Model must not be an empty String'	As expected
Type of server is 'JBoss'. No dataSourceJndiName is specified	No errors	As expected
Type of server is 'JBoss'. Also, a dataSourceJndiName is specified	No errors	As expected
Type of server is 'webLogic6.1' and a dataSourceJndiName is specified	No errors	As expected

Test: If primary key is auto-generated by 3E, the type of the primary key field must be java.lang.Long, java.lang.Integer or java.lang.String

Input data	Expected result	Test result
A primary key is specified to be auto-generated by 3E and the type is specified to be java.lang.Double	IllegalInputException: 'The system only supports autogeneration of primary key of type: java.lang.String, java.lang.Integer and java.lang.Long'	As expected
All Primary keys auto-generated by 3E are specified to be java.lang.Long, java.lang.Integer or java.lang.String	No errors	As expected

Test: The type of a CMR collection fields must always be java.util.Collection or java.util.Set

Input data	Expected result	Test result
A CMR collection field in relation Cruise_Reservation is specified to be 'java.util.ArrayList'	IllegalInputException: 'In Cruise_Reservatioin: The type of a cmrField in a many relation can only be java.util.Collection or java.util.Set'	As expected

...continues on next page...

A CMR collection field in relation Cruise_Reservation is specified to be 'Collection'	IllegalInputException: 'In Cruise_Reservatioin: The type of a cmrField in a many relation can only be java.util.Collection or java.util.Set'	As expected
A CMR collection field in relation Cruise_Reservation is specified to be 'Set'	IllegalInputException: 'In Cruise_Reservatioin: The type of a cmrField in a many relation can only be java.util.Collection or java.util.Set'	As expected
A CMR collection field in relation Cruise_Reservation is specified with no type	No errors	As expected
The type of all CMR collection fields are specified to be java.util.Collection or java.util.Set	No errors	As expected

Test: An Entity Bean must not have two or more fields with the same name

Input data	Expected result	Test result
An Entity Bean 'Cabin' with two fields named 'ship' is specified	IllegalInputException: 'Cabin has two Fields with name ship'	As expected
An Entity Bean 'Cabin' with three fields named 'ship' is specified	IllegalInputException: 'Cabin has two Fields with name ship'	As expected
Two Entity Beans are specified, each of them have a field with name 'ship'	No errors	As expected
None of the specified Entity Beans have two or more fields with the same name belonging to the same Entity Bean	No errors	As expected

Test: Fields to be initialized by a create method must always match a field in the Entity Bean

Input data	Expected result	Test result
In Entity Bean 'Cabin' a create method is specified to initialize a field called 'tonnage'. Cabin has no field called 'tonnage'	IllegalInputException: 'Field tonnage to instantiate in CreateMethod must match a field in the EntityEjb Cabin'	As expected

...continues on next page...

In Entity Bean 'Cabin' a create method is specified to initialize a field called 'tonnage' and one called 'reservation'. Cabin has no fields called 'tonnage' and 'reservation'	IllegalInputException: 'Field tonnage to instantiate in CreateMethod must match a field in the EntityEjb Cabin'	As expected
All specified create methods initialize fields matching fields in the Entity Bean	No errors	As expected

Test: If the primary key is auto-generated by 3E, the primary key field must not be among the fields initialized by a create method

Input data	Expected result	Test result
In Entity Bean 'Cabin a create method is specified to initialize a field called 'id' matching the auto-generated primary key field 'id'	IllegalInputException: 'Field id to instantiate in CreateMethod must not match the cmpPkField in the EntityEjb Cabin since primary key is automatically generated and need not be instantiated'	As expected
None of the specified create methods initialize fields matching an auto-generated primary key field in the Entity Bean	No errors	As expected

Appendix D

The XML input file for the Titan business test case

In this appendix the XML input file for the Titan business test case is shown.

Appendix E

JSP clients used in the Titan business test case

The following tables contains descriptions of JSP clients used in the Titan business test case and the outcome of executing each client. Note that all clients have been executed on both JBoss and WebLogic6.1 J2EE application servers. All JSP client scripts can be found at `http://www.it-c.dk/people/anne/3E`

JSP clients demonstrating and testing simple relationships centered on the Customer EJB

Description of JSP-client	Expected result	Test result
ex71_a.jsp: Tests the one-to-one bidirectional relationship between Customer and CreditCard. First a Customer EJB and a CreditCard EJB are created. Then they are linked together and both directions in the relationship are tested. Finally the two EJBs are unlinked	When the two EJBs are linked together, the database should contain corresponding references between the Customer and the CreditCard. After the EJBs are unlinked, the references should be gone on both sides	As expected
ex71_b.jsp: Tests the one-to-one unidirectional relationship between Customer and Address. First a reference to the Customer created in ex71_a.jsp is obtained. This Customer has no address. An Address EJB is created and attached to the Customer. The Address EJB is then modified through the Customer's address reference. Finally another Address EJB is created and this is attached to the Customer	At first the database should reflect that the Customer has no address. Second an address reference should be present and modifications done to the Address EJB should be reflected in the database. Finally, when a new Address EJB is attached to the Customer, this should effectively orphans the first Address EJB and Customer should only be linked to the new Address EJB	As expected

...continues on next page...

Description of JSP-client	Expected result	Test result
ex71_c.jsp: Tests the one-to-many unidirectional relationship between Customer and Phone. First the Customer from previous is located. A new Phone EJB is added to the Customers phone collection. An additional Phone EJB of a different type than the first one is added to the Customers phone collection. The first Phone EJB is modified through the Customer, before it is finally removed from the Customers phone collection. Everything is done using business-methods declared on the Customer	As in previous examples all references between Customer and Phone EJBs and all modifications must at all times be represented correctly in the database. The final result should be that the second Phone EJB is still attached to the Customer, while the first Phone EJB is left orphaned	As expected

JSP clients demonstrating the remaining four relationship types: many-to-one unidirectional (Cruise-ship), one-to-many bidirectional (Cruise-Reservation), many-to-many bidirectional (Customer-Reservation) and many-to-many unidirectional (Cabin-Reservation)

Description of JSP-client	Expected result	Test result
ex72_a.jsp: Tests the many-to-one unidirectional relationship between Cruise and Ship. Creates Ship A, Ship B and Cruise 1-2-3-4-5-6. Links Cruise 1-2-3 to Ship A and Cruise 4-5-6 to Ship B. Then Cruise 1 is changed to use same ship as Cruise 4	The database should at all times reflect the corresponding references between Cruise EJBs and Ship EJBs. The final result should be that Cruise 1-4-5-6 are linked to Ship B and Cruise 2-3 are linked to Ship A	As expected
ex72_b.jsp: Tests the one-to-many bidirectional relationship between Cruise and Reservation, including the use of set methods to modify the reservations associated with a Cruise. Creates Cruise A, Cruise B and Reservation 1-2-3-4-5-6. When Reservation EJBs are created, a Cruise EJB is included in the input parameters, thereby linking a Reservation EJB to a Cruise EJB from the start. Reservation 1-2-3 are linked to Cruise A, Reservartion 4-5-6 to Cruise B. Finally Reservations linked to Cruise A is are passed to Cruise B using the get- and setReservation methods respectively	The final result should be that Reservation 1-2-3 originally linked to Cruise A are now linked to Cruise B. Reservation 4-5-6 originally linked to Cruise B are orphaned	As expected

...continues on next page...

Description of JSP-client	Expected result	Test result
ex72_c.jsp: Tests the one-to-many bidirectional relationship between Cruise and Reservation like in ex72_b.jsp, using the business method `addAll` to modify reservations associated with a cruise	All Reservation EJBs should be linked to Cruise B, while Cruise A has no reservations	As expected
ex72_d.jsp: Tests the many-to-many bidirectional relationship between Customer and Reservation. Creates a Ship and a Cruise, six Customers and two Reservations. Reservation 1 includes Customer 1-2-3, while Reservation 2 includes Customer 4-5-6. Then customers linked to Reservation 2 is added to Reservation 1 using a business method	The final result should be that all six customers are linked to Reservation 1, while customer 4-5-6 is also linked to Reservation 2	As expected
ex72_e.jsp: Tests the many-to-many bidirectional relationship between Customer and Reservation like in ex72_d.jsp, using set methods to modify customers associated to reservations. Creates Customers 1-2-3-4-5-6 and Reservation 1-2-3-4. Links three customers to each reservation. Modify Reservation 4 to match Reservation 1	The result should be that all reservations includes three customers. Reservation 1 and 4 should include the same customers. Customer 6 should not be included in any reservation	As expected
ex72_f.jsp: Tests the many-to-many unidirectional relationship between Cabin and Reservation, including the removal of cabins from a reservation using an iterator. Creates Cabin 1-2-3-4-5-6 and Reservation 1-2-3-4. Links three cabins to each reservations. Then all cabins are removed from Reservation 1 using an iterator to iterate through the reservation's collection of cabins	Reservation 2-3-4 should each include three cabins, while Reservation 1 has no cabins	As expected
ex72_g.jsp: Tests the many-to-many unidirectional relationship between Cabin and Reservation using set methods to modify cabins associated with reservations. Cabin 1-2-3 and Reservation 1-2 is created. Cabin 1-2 is linked to Reservation 1 and Cabin 2-3 is linked to Reservation 2. The appropriate set methods are used to link cabins associated with Reservation 1 to Reservation 2	Both reservations should finally be associated to Cabin 1 and 2, while Cabin 3 is not included in any reservation	As expected

JSP clients exploring some basic EJB QL, including finder methods, select methods and the use of the IN operation in queries

Description of JSP-client	Expected result	Test result
ex81_a.jsp: Creates a set of Customer EJBs and tests the `findByName` finder method in the Customer home interface	Should find Customer John Smith85, 1085 Elm Street Eagan, CA 55401	As expected
ex81_b.jsp: Tests the `findByCity` finder method in the Customer home interface	Should find Customer John Smith80, John Smith86, John Smith92 and John Smith98, all living in Minneapolis	As expected
ex81_c.jsp: Tests the `ejbSelectZipCodes` select method in the Address Bean-Class	Should find zip codes 55401, 55403, 55405, 55402 and 55404	As expected
ex81_d.jsp: Creates a Ship, a Cruise and six Customers to be used in the subsequent JSP clients. Each customer has two reservations and each reservation is for two cabins	All created EJBs including relationship references should be present i the database	As expected
ex81_e.jsp: Tests the `findAllOnDeckLevel` finder method in the Cabin home interface	Should find Cabin 3001, Cabin 3002, Cabin 3011 and Cabin 3012, all on deck 3	As expected
ex81_f.jsp: Tests the `ejbSelectAllForCustomer` select method in the Cabin Bean-Class	Should find Cabin 1012, Cabin 1011, Cabin 1001 and Cabin 1002, all included in reservations for Customer 1	As expected

JSP clients exploring some more complex operations and comparison operators available in EJB QL

Description of JSP-client	Expected result	Test result
ex82_a.jsp: Tests the `findByTonnage` methods of the Ship home interface. Creates 10 ships with various tonnage values. Then asks for ships with exactly 100K tonnage and for ships with tonnage between 50K and 110K	Should find only Ship 7 to have exactly 100K tonnage and ships 2-3-4-5-6-7-8 to have tonnage between 50K and 110K	As expected

...continues on next page...

Description of JSP-client	Expected result	Test result
ex82_b.jsp: Just performs Customer setup for subsequent JSP clients	As a final result the database should contain a fairly large set of Customer EJBs with associated Address, Phone, Cruise and Reservation EJBs	As expected
ex82_c.jsp: Tests the `findByExactName` and the `findByName` finder methods in the Customer home interface. Is asked to find 1) customers having name exactly matching 'Joe Star', 2) customers having a name like 'Jo% S%' and 3) like 2, but the customer should also live in Minneapolis (Note: % represents a wildcard value)	Should find: 1) one customer with name 'Joe Star', 2) six customers with name 'Jo% S%' and 3) three customers from result in 2 do also live in Minneapolis	As expected
ex82_d.jsp: Tests the `findInHotStates` method of the Customer home interface	Should find 10 customers all living in hot states	As expected
ex82_e.jsp: Tests the `findWithNoReservation` and `findOnCruise` methods of the Customer home interface. Is asked to find 1) Customers with no reservations and 2) Customers with reservations on Alaska Cruise	Should find: 1) Customers 81-84-87-90-93-96-99 and 2) Customers 82-85-88-91-94-97	As expected
ex82_f.jsp: Tests the `findByState` method in the Customer home interface	Should find 10 customers living in Minneapolis	As expected

Appendix F

3E source code

XML Schemas: 5 pages

ejbSchema4_0.xsd.....p.1-4
connectionSchema1_0.xsd.....p.5

Mapping files: 8 pages

EntityEjbMapping.xml..... p.1-2
MethodMapping.xml..... p.3-5
ModelMapping.xml..... p.6
RelationMapping.xml..... p.7
ConnectionMapping.xml.....p.8

Java source code: 79 pages

dk.itu.bogm.generate.parsers.BogmUnmarshaller.java.....p.1-2
dk.itu.bogm.generate.templateUtil.Utility.java..... p.2-3
dk.itu.bogm.db.util.ConnectionSettings.java..... p.4-5
dk.itu.bogm.db.util.Sql2Java.java..... p.5-8
dk.itu.bogm.db.util.DbStringUtil.java.....p.8-9
dk.itu.bogm.db.db2xml.Relation.java..... p.9-13
dk.itu.bogm.db.db2xml.Table.java..... p.14-15
dk.itu.bogm.db.db2xml.SchemaFactory.java..... p.15-18
dk.itu.bogm.db.db2xml.Schema.java..... p.18-19
dk.itu.bogm.db.db2xml.Many2ManyRelation.java.....p.20
dk.itu.bogm.common.util.StringUtil.java..... p.21
dk.itu.bogm.common.exceptions.InternalErrorException.java..... p.21
dk.itu.bogm.common.exceptions.VisitorErrorException.java..... p.22
dk.itu.bogm.common.exceptions.ExternalErrorException.java.....p.22
dk.itu.bogm.common.exceptions.IllegalInputException.java..... p.23
dk.itu.bogm.common.exceptions.NoPrimaryKeyException.java.....p.23
dk.itu.bogm.build.visitors.BuildModelVisitor.java..... p.24-26
dk.itu.bogm.build.visitors.DataCheckVisitor.java..... p.26-30
dk.itu.bogm.build.visitors.LegalDataIIVisitor.java..... p.30-32
dk.itu.bogm.build.visitors.LegalDataVisitor.java..... p.32-34
dk.itu.bogm.build.visitors.SetParentVisitor.java..... p.34-37
dk.itu.bogm.build.visitors.AbstractVisitor.java..... p.37-41

dk.itu.bogm.model.EjbCreateMethod.java.....p.41-42
dk.itu.bogm.model.BeanClass.java..... p.43-45
dk.itu.bogm.model.Deployment.java..... p.45-46
dk.itu.bogm.model.BusinessMethod.java..... p.46
dk.itu.bogm.model.CmpField.java.....p.47
dk.itu.bogm.model.CmpPkField.java.....p.47-48
dk.itu.bogm.model.CmrField.java..... p.48
dk.itu.bogm.model.CreateMethod.java..... p.49
dk.itu.bogm.model.PrivateMethod.java..... p.50
dk.itu.bogm.model.EntityEjb.java..... p.51-55
dk.itu.bogm.model.Field.java.....p.56-57
dk.itu.bogm.model.Method.java.....p.58-59
dk.itu.bogm.model.EjbPostCreateMethod.java.....p.59-60
dk.itu.bogm.model.FindMethod.java..... p.60-61
dk.itu.bogm.model.HomeMethod.java.....p.61-62
dk.itu.bogm.model.LocalHomeInterface.java.....p.62-63
dk.itu.bogm.model.SelectMethod.java..... p.64
dk.itu.bogm.model.Relation.java..... p.65-66
dk.itu.bogm.model.Model.java..... p.66-67
dk.itu.bogm.model.Parameter.java..... p.68
dk.itu.bogm.model.LocalComponentInterface.java..... p.69-70
dk.itu.bogm.model.Named.java..... p.70
dk.itu.bogm.model.MethodBody.java.....p.71
dk.itu.bogm.model.Origin.java..... p.72
dk.itu.bogm.model.Query.java..... p.73
dk.itu.bogm.model.Role.java..... p.74-75
dk.itu.bogm.Xml2ejb.java..... p.76-77
dk.itu.bogm.Db2xml.java..... p.77-79

Velocity templates: 18 pages

ejb-jar-xmlTemplate.vm..... p.1-4
BeanClassTemplate.vm..... p.5-7
VM_global_library.vm.....p.8
generate.vm.....p.9
LocalComponentInterfaceTemplate.vm..... p.10
LocalHomeInterfaceTemplate.vm..... p.11
primaryKeyClassTemplate.vm..... p.12-13
GUIDProviderTemplate.vm.....p.14-15
weblogic-ejb-jar-xml.vm.....p.16
weblogic-cmp-rdbms-jar-xml.vm..... p.17-18